Talos Vulnerability Report

TALOS-2021-1250

Microsoft Azure Sphere mqueue inode initialization kernel code execution vulnerability

APRIL 13, 2021

CVE NUMBER

CVE-2021-27080

Summary

A code execution vulnerability exists in the mqueue inode initialization functionality of Microsoft Azure Sphere 21.01. A specially crafted set of syscalls can lead to uninitialized kernel read, which in turn leads to code execution in kernel. To trigger this vulnerability, an attacker can either create and open an mqueue in the root IPC namespace, or just create and destroy an IPC namespace.

Tested Versions

Microsoft Azure Sphere 21.01

Product URLs

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

CVSSv3 Score

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

CWE

CWE-457 - Use of Uninitialized Variable

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.

Before beginning, it's important to note that triggering this vulnerability is not dependent on the `clone()` or `unshare()` syscalls, it's also possible to trigger it simply by creating and opening an mqueue in the root IPC namespace with a really large `mq_maxsize` or `mq_msgsize`. For this report, we go indepth into the `unshare` code path since it is easier to trigger, but everything written here also applies to the creation of the root IPC namespace.

A Linux namespace is an abstraction provided by the kernel to limit the execution context of a given process or thread and also potentially to isolate it.
Currently, there exist 8 kinds of namespaces: Cgroup, IPC, Network, Mount, PID, Time, User, UTS. An unprivileged user can create a new user namespace (using the `CLONE_NEWUSER` flag) and have a full capabilities (root user with all caps) in that namespace. From `man user_namespaces(7)`:

```
User namespaces isolate security-related identifiers and
attributes, in particular, user IDs and group IDs (see
credentials(7)), the root directory, keys (see keyrings(7)), and
capabilities (see capabilities(7)).  A process's user and group
IDs can be different inside and outside a user namespace.  In
particular, a process can have a normal unprivileged user ID
outside a user namespace while at the same time having a user ID
of 0 inside the namespace; in other words, the process has full
privileges for operations inside the user namespace, but is
unprivileged for operations outside the namespace.
```

Once in the new user namespace, the user has the `CAP_SYS_ADMIN` capability in the namespace. This means that it's possible to use the `CLONE_NEWIPC` flag to switch to a new IPC namespace. From `man ipc_namespaces(7)`:

```
IPC namespaces isolate certain IPC resources, namely, System V
IPC objects (see sysvipc(7)) and (since Linux 2.6.30) POSIX
message queues (see mq_overview(7)).  The common characteristic
of these IPC mechanisms is that IPC objects are identified by
mechanisms other than filesystem pathnames.

[...]

The following /proc interfaces are distinct in each IPC namespace:

*  The POSIX message queue interfaces in /proc/sys/fs/mqueue.
[...]

When an IPC namespace is destroyed (i.e., when the last process
that is a member of the namespace terminates), all IPC objects in
the namespace are automatically destroyed.                         // [1]
```

The part of import that we most care about is the last line at [1], "When an IPC namespace is destroyed… all IPC objects in the namespace are automatically destroyed". It's rather clear in general, but let us examine what this means materially. For each IPC namespace, there exists a `struct mqueue_fs_context`:

```
static int mqueue_init_fs_context(struct fs_context *fc)
{
    struct mqueue_fs_context *ctx;

    ctx = kzalloc(sizeof(struct mqueue_fs_context), GFP_KERNEL);
    if (!ctx)
        return -ENOMEM;

    ctx->ipc_ns = get_ipc_ns(current->nsproxy->ipc_ns);
    put_user_ns(fc->user_ns);
    fc->user_ns = get_user_ns(ctx->ipc_ns->user_ns);
    fc->fs_private = ctx;
    fc->ops = &mqueue_fs_context_ops;
    return 0;
}
```

This filesystem driver holds all the information about the mqueues of the current IPC namespace and so, like any other filesystem, there must be a superblock:

```
static int mqueue_fill_super(struct super_block *sb, struct fs_context *fc)
{
    struct inode *inode;
    struct ipc_namespace *ns = sb->s_fs_info;

    sb->s_iflags |= SB_I_NOEXEC | SB_I_NODEV;
    sb->s_blocksize = PAGE_SIZE;
    sb->s_blocksize_bits = PAGE_SHIFT;
    sb->s_magic = MQUEUE_MAGIC;
    sb->s_op = &mqueue_super_ops;

    inode = mqueue_get_inode(sb, ns, S_IFDIR | S_ISVTX | S_IRWXUGO, NULL); // [1]
    if (IS_ERR(inode))
        return PTR_ERR(inode);

    sb->s_root = d_make_root(inode);
    if (!sb->s_root)
        return -ENOMEM;
    return 0;
}
```

The only point of interest for us is the `mqueue_get_inode` call at [1], as the superblock inode will be important later.

```
static struct inode *mqueue_get_inode(struct super_block *sb,
        struct ipc_namespace *ipc_ns, umode_t mode,
        struct mq_attr *attr)
{
    struct user_struct *u = current_user();
    struct inode *inode;
    int ret = -ENOMEM;

    inode = new_inode(sb);
    if (!inode)
        goto err;

    inode->i_ino = get_next_ino();
    inode->i_mode = mode;
    inode->i_uid = current_fsuid();
    inode->i_gid = current_fsgid();
    inode->i_mtime = inode->i_ctime = inode->i_atime = current_time(inode);

    if (S_ISREG(mode)) { // mode == S_IFDIR | S_ISVTX | S_IRWXUGO
    //[...]

    } else if (S_ISDIR(mode)) {        // [1]
        inc_nlink(inode);
        /* Some things misbehave if size == 0 on a directory */
        inode->i_size = 2 * DIRENT_SIZE;
        inode->i_op = &mqueue_dir_inode_operations;
        inode->i_fop = &simple_dir_operations;
    }

    return inode;
out_inode:
    iput(inode);
err:
    return ERR_PTR(ret);
}
```

Since our inode is in fact a directory (mode => `S_IFDIR | S_ISVTX | S_IRWXUGO`), aside from the common inode initialization, we only hit the branch at [1], which sets `inode->i_size`, `inode->i_op`, and `inode->i_fop`. Important to note: there is nothing else that gets initialized in this function. Returning back to our quote of import: "When an IPC namespace is destroyed… all IPC objects in the namespace are automatically destroyed", we now examine exactly how our IPC namespace's mqueue filesystem's superblock's root dentry inode is destroyed, starting with `mq_put_mnt` for context:

```
void mq_put_mnt(struct ipc_namespace *ns)
{
    kern_unmount(ns->mq_mnt);
}
```

A simple function, yet necessary to bring up. We reach `mq_put_mnt` when a given IPC namespace is destroyed (i.e. there are no more processes which utilize the given IPC namespace). Logically, to destroy the IPC namespace, it is necessary to unmount the mqueue filesystem that was created. We now skip a lot of the following backtrace and go straight to `shrink_dcache_for_umount`:

```
#0  shrink_dcache_for_umount (sb=0xc138b800) at fs/dcache.c:1621
#1  0xc01b3e08 in generic_shutdown_super (sb=0xc138b800) at fs/super.c:447
#2  0xc01b3eee in kill_anon_super (sb=<optimized out>) at fs/super.c:1108
#3  0xc01b42c2 in deactivate_locked_super (s=0xc138b800) at fs/super.c:335
#4  0xc01b4332 in deactivate_super (s=<optimized out>) at fs/super.c:366
#5  0xc01c4e88 in cleanup_mnt (mnt=0xc1c52f00) at fs/namespace.c:1102
#6  0xc01c4fb4 in mntput_no_expire (mnt=0xc138b800) at fs/namespace.c:1185
#7  0xc01c4ff6 in mntput (mnt=<optimized out>) at fs/namespace.c:1195
#8  0xc01c51c4 in kern_unmount (mnt=<optimized out>) at fs/namespace.c:3809
#9  0xc023a0aa in mq_put_mnt (ns=<optimized out>) at ipc/mqueue.c:1645
#10 0xc023a43c in put_ipc_ns (ns=0xc1c05000) at ipc/namespace.c:151
#11 0xc0125ab0 in free_nsproxy (ns=0xc13c1200) at kernel/nsproxy.c:187
#12 0xc0125b6e in switch_task_namespaces (p=<optimized out>, new=<optimized out>) at kernel/nsproxy.c:240
#13 0xc0125b7c in exit_task_namespaces (p=<optimized out>) at kernel/nsproxy.c:245
#14 0xc0115be6 in do_exit (code=<optimized out>) at kernel/exit.c:819
#15 0xc0115fc4 in do_group_exit (exit_code=-1052995584) at kernel/exit.c:918
#16 0xc0116002 in __do_sys_exit_group (error_code=<optimized out>) at kernel/exit.c:929
#17 __se_sys_exit_group (error_code=<optimized out>) at kernel/exit.c:927
#18 <signal handler called>

/*
 * destroy the dentries attached to a superblock on unmounting
 */
void shrink_dcache_for_umount(struct super_block *sb)
{
    struct dentry *dentry;

    WARN(down_read_trylock(&sb->s_umount), "s_umount should've been locked");

    dentry = sb->s_root;
    sb->s_root = NULL;
    do_one_tree(dentry); // [1]

    while (!hlist_bl_empty(&sb->s_roots)) {
        dentry = dget(hlist_bl_entry(hlist_bl_first(&sb->s_roots), struct dentry, d_hash));
        do_one_tree(dentry);
    }
}
```

Again, simple yet needed, before totally unmounting a given filesystem, it is necessary to destroy all the objects that the kernel is using to track the data. Eventually once this is done, we must also destroy the filesystem's superblock's root dentry, which is done in `do_one_tree` at [1]:

```
static void do_one_tree(struct dentry *dentry)
{
    shrink_dcache_parent(dentry);
    d_walk(dentry, dentry, umount_check);
    d_drop(dentry);
    dput(dentry); // [1]
}
```

Most important, we call `dput` on our dentry, which eventually leads us down the following code path:

```
#15 0xc01c26a4 in iput_final (inode=<optimized out>) at fs/inode.c:1573
#16 iput (inode=0xc1392400) at fs/inode.c:1602  // from dentry_unlink_inode...
#17 0xc01bff06 in __dentry_kill (dentry=0xc18afb28) at fs/dcache.c:586
#18 0xc01bff88 in dentry_kill (dentry=<optimized out>) at fs/dcache.c:693
#19 dput (dentry=0x0) at fs/dcache.c:866
#20 0xc01c080c in do_one_tree (dentry=<optimized out>) at fs/dcache.c:1611
```

Once we get to `iput_final`, we're almost back to filesystem-specific code, instead of Linux code, but let us walk through regardless:

```
/*
 * Called when we're dropping the last reference
 * to an inode.
 *
 * Call the FS "drop_inode()" function, defaulting to
 * the legacy UNIX filesystem behaviour.  If it tells
 * us to evict inode, do so.  Otherwise, retain inode
 * in cache if fs is alive, sync and evict if fs is
 * shutting down.
 */
static void iput_final(struct inode *inode)
{
    struct super_block *sb = inode->i_sb;
    const struct super_operations *op = inode->i_sb->s_op;
    int drop;

    WARN_ON(inode->i_state & I_NEW);

    if (op->drop_inode)
        drop = op->drop_inode(inode);
    else
        drop = generic_drop_inode(inode); // [1]

    if (!drop && (sb->s_flags & SB_ACTIVE)) { // skip
            inode_add_lru(inode);
            spin_unlock(&inode->i_lock);
            return;
    }

    if (!drop) { // skip
        //[...]
    }

    inode->i_state |= I_FREEING;
    if (!list_empty(&inode->i_lru)) // [2]
        inode_lru_list_del(inode);
    spin_unlock(&inode->i_lock);

    evict(inode);
}
```

At [1], because the mqueue file system does not have its own `.drop_inode` method, we instead assign the `generic_drop_inode(inode)` at [1], which means we skip both of the conditionals checking for `!drop`, and end up at the call to `list_empty` [2]:

```
/**
 * list_empty - tests whether a list is empty
 * @head: the list to test.
 */
static inline int list_empty(const struct list_head *head)
{
    return READ_ONCE(head->next) == head;
}
```

The `list_empty` function determines that a list is empty if the `head->next` member of a list points to itself. For a concrete example, the following `inode->i_lru` list would be considered empty:

```
[-.-]> p/x ((struct inode *)0xc1a02278)->i_lru
$6 = {next = 0xc1a0231c, prev = 0xc1a0231c}

[-.-]> p/x &((struct inode *)0xc1a02278)->i_lru
$7 = 0xc1a0231c
```

It's considered empty not because `list->next == list->prev`, but because `&list == list->next`. Examining the state of our mqueue superblock dentry inode's `i_lru` list at this point:

```
[>_>]> p/x ((struct inode *)0xc13acc00)->i_lru
$8 = {next = 0x0, prev = 0x0}

[<_<]> p/x &((struct inode *)0xc13acc00)->i_lru
$9 = 0xc13acca4
```

Curiously, `i_lru->next` == 0x0, which means that `list_empty` does not consider the mqueue's `i_lru` list to be empty (since `&i_lru` != 0x0). Thus, we end up hitting `inode_lru_list_del` with a list pointing to 0x0:

```
    if (!list_empty(&inode->i_lru))
        inode_lru_list_del(inode);
```

In order to see why this occurs, we first track down exactly where the mqueue inode is allocated:

```
static struct inode *mqueue_alloc_inode(struct super_block *sb)
{
    struct mqueue_inode_info *ei;

#ifdef CONFIG_DISABLE_MQUEUE_INODE_CACHE
    ei = kmalloc(sizeof(struct mqueue_inode_info), GFP_KERNEL);  // [1]
#else
    ei = kmem_cache_alloc(mqueue_inode_cachep, GFP_KERNEL);      // [2]
#endif /* CONFIG_DISABLE_MQUEUE_INODE_CACHE */
    if (!ei)
        return NULL;

    return &ei->vfs_inode;
}
```

Because of an optimization define (`CONFIG_DISABLE_MQUEUE_INODE_CACHE`), instead of using a named `kmem_cache` for mqueue inodes [2], they are generically allocated via `kmalloc` [1]. However it's important to note that, the lines at [1] and [2] are not equivalent (ignoring the obvious effect of which `kmem_cache` the object's slab belongs to). Looking at the initialization of `mqueue_inode_cache` shows us why:

```
static int __init init_mqueue_fs(void)
{
    int error;

#ifndef CONFIG_DISABLE_MQUEUE_INODE_CACHE
    mqueue_inode_cachep = kmem_cache_create("mqueue_inode_cache",
                                            sizeof(struct mqueue_inode_info), 0,
                                            SLAB_HWCACHE_ALIGN|SLAB_ACCOUNT, init_once); // [1]
    if (mqueue_inode_cachep == NULL)
        return -ENOMEM;
#endif /* CONFIG_DISABLE_MQUEUE_INODE_CACHE */
//[...]
```

Aside from just the name of the `kmem_cache` and the size of the objects therein, there's also a few more arguments to `kmeme_cache_create`, most importantly the last argument `init_once` at [1]:

```
/**
 * kmem_cache_create - Create a cache.
 * @name: A string which is used in /proc/slabinfo to identify this cache.
 * @size: The size of objects to be created in this cache.
 * @align: The required alignment for the objects.
 * @flags: SLAB flags
 * @ctor: A constructor for the objects. //[1]
```

According to the comments in `mm/slab_common.c` for `kmeme_cache_create`, this `init_once` argument is apparently a constructor for the allocated mqueue inodes. Following the code path to see what this actually does:

```
#ifndef CONFIG_DISABLE_MQUEUE_INODE_CACHE
static void init_once(void *foo)
{
    struct mqueue_inode_info *p = (struct mqueue_inode_info *) foo;

    inode_init_once(&p->vfs_inode);
}
#endif /* CONFIG_DISABLE_MQUEUE_INODE_CACHE */
```

Which then leads us to `inode_init_once`:

```
/*
 * These are initializations that only need to be done
 * once, because the fields are idempotent across use
 * of the inode, so let the slab aware of that.
 */
void inode_init_once(struct inode *inode)
{
    memset(inode, 0, sizeof(*inode));
    INIT_HLIST_NODE(&inode->i_hash);
    INIT_LIST_HEAD(&inode->i_devices);
    INIT_LIST_HEAD(&inode->i_io_list);
    INIT_LIST_HEAD(&inode->i_wb_list);
    INIT_LIST_HEAD(&inode->i_lru);  //[1]
    __address_space_init_once(&inode->i_data);
    i_size_ordered_init(inode);
}
```

Among the numerous and generic inode member initializations that must occur, we can see our `inode->i_lru` member being initialized at [1]. Thus, because the allocated mqueue inodes within `mqueue_alloc_inode` are not initialized by `inode_init_once` after being `kmalloc()`'ed, an attacker is able to poison these uninitialized members of the inode (via a kernel heap spray and free before allocation of the mqueue's inode), resulting in code execution.

Finally, it's also worth noting that, to trigger this vulnerability, one must simply enter a new user namespace with a new IPC namespace (`unshare -i`), and then exit that namespace (`exit`).

Crash Information

In the following crash, note that we control the address of `i_lru` (via kernel heap poisioning)

```
[    8.075465] 8<--- cut here ---
[    8.077510] Unable to handle kernel paging request at virtual address 41414145
[    8.078364] pgd = (ptrval)
[    8.078754] [41414145] *pgd=00000000
[    8.080888] Internal error: Oops: 805 [#1] THUMB2
[    8.081363] Modules linked in:
[    8.082046] CPU: 0 PID: 74 Comm: test.out Not tainted 5.4.54-mt3620-azure-sphere+ #3
[    8.082522] Hardware name: Generic DT based system
[    8.083879] PC is at list_lru_del+0x1e/0x3a
[    8.084361] LR is at list_lru_del+0x1b/0x3a
[    8.084677] pc : [<c019858a>]    lr : [<c0198587>]    psr: 60000033
[    8.085002] sp : c13d0ea8  ip : c13d0ebc  fp : 00000000
[    8.085324] r10: c0f79f04  r9 : c0f79c00  r8 : c13d0000
[    8.085785] r7 : c0701c48  r6 : c0f79e58  r5 : c13a6380  r4 : c0f798a4
[    8.086273] r3 : 41414141  r2 : 41414141  r1 : c0f798a4  r0 : c13a6380
[    8.086717] Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA Thumb  Segment user
[    8.087394] Control: 50c5387d  Table: 413a0059  DAC: 00000055
[    8.087935] Process test.out (pid: 74, stack limit = 0x(ptrval))
[    8.088560] Stack: (0xc13d0ea8 to 0xc13d1000)
[    8.089008] 0ea0:                   c0f79c00 c0f79800 c13d0ec4 c01c17c5 00000120 c01c2345
[    8.089559] 0ec0: c0f79c00 c13d0ec4 c13d0ec4 33fe9b86 c0f79c00 c0414640 c13d0000 c13d0f34
[    8.090263] 0ee0: c0701c48 c13d44ac 000000f8 c01b3e1d 00000000 00000010 c072e964 c01b3eef
[    8.090743] 0f00: c0f79c00 c01b42c3 33fe9b86 c13d4480 00000000 c01c4e89 c13d4480 c0704a58
[    8.091695] 0f20: c13d0000 c13d0f34 c0701c48 c01c4fb5 c13d4480 c13d0f34 c13d0f34 33fe9b86
[    8.092415] 0f40: c1c05600 c0704a58 c13d0000 c1380738 fffff000 00000001 000000f8 c023a43d
[    8.092872] 0f60: c13a66c0 c0125ab1 00000000 c13ce000 c1380700 c0115be7 00000000 fffff000
[    8.093183] 0f80: c13ce244 33fe9b86 00000000 000000f8 00000000 c0115fc5 00000000 c0116003
[    8.093503] 0fa0: 00000000 c0101001 00000000 bef83ee4 00000000 bef83e10 880ec981 00000000
[    8.094025] 0fc0: 00000000 bef83ee4 00000001 000000f8 00000001 00000000 00000000 00000000
[    8.094734] 0fe0: 88076c0c bef83ea8 8808502f 880b33c2 60000030 00000000 00000000 00000000
[    8.096464] [<c019858a>] (list_lru_del) from [<c01c17c5>] (inode_lru_list_del+0x17/0x32)
[    8.097301] [<c01c17c5>] (inode_lru_list_del) from [<c01c2345>] (evict_inodes+0x51/0xa4)
[    8.097840] [<c01c2345>] (evict_inodes) from [<c01b3e1d>] (generic_shutdown_super+0x29/0x94)
[    8.098383] [<c01b3e1d>] (generic_shutdown_super) from [<c01b3eef>] (kill_anon_super+0xf/0x18)
[    8.098843] [<c01b3eef>] (kill_anon_super) from [<c01b42c3>] (deactivate_locked_super+0x3b/0x62)
[    8.099475] [<c01b42c3>] (deactivate_locked_super) from [<c01c4e89>] (cleanup_mnt+0x75/0x98)
[    8.099885] [<c01c4e89>] (cleanup_mnt) from [<c01c4fb5>] (mntput_no_expire+0x109/0x130)
[    8.100308] [<c01c4fb5>] (mntput_no_expire) from [<c023a43d>] (put_ipc_ns+0x1f/0x4a)
[    8.100747] [<c023a43d>] (put_ipc_ns) from [<c0125ab1>] (free_nsproxy+0x2d/0x44)
[    8.101123] [<c0125ab1>] (free_nsproxy) from [<c0115be7>] (do_exit+0x287/0x5fc)
[    8.101504] [<c0115be7>] (do_exit) from [<c0115fc5>] (do_group_exit+0x41/0x6c)
[    8.102050] [<c0115fc5>] (do_group_exit) from [<c0116003>] (__wake_up_parent+0x1/0x1c)
[    8.102871] Code: f7ff ff5d e894 000c (6053) 601a
[    8.103913] ---[ end trace e7290cc550e9085a ]---
[    8.104254] Kernel panic - not syncing: Fatal exception
[    9.106127] Reboot failed -- System halted
```

Mitigation

In file `ipc/mqueue.c`:

```
#endif /* CONFIG_DISABLE_MQUEUE_INODE_CACHE */
static struct inode *mqueue_alloc_inode(struct super_block *sb)
{
    struct mqueue_inode_info *ei;

#ifdef CONFIG_DISABLE_MQUEUE_INODE_CACHE
    ei = kmalloc(sizeof(struct mqueue_inode_info), GFP_KERNEL);
    inode_init_once(&ei->vfs_inode);  // <-------------------------------------------- add this
#else
    ei = kmem_cache_alloc(mqueue_inode_cachep, GFP_KERNEL);
#endif /* CONFIG_DISABLE_MQUEUE_INODE_CACHE */
    if (!ei)
        return NULL;

    return &ei->vfs_inode;
}
```

Timeline

2021-02-16 - Vendor Disclosure
2021-04-13 - Public Release

CREDIT

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

VULNERABILITY REPORTS

PREVIOUS REPORT
TALOS-2021-1247

NEXT REPORT
TALOS-2021-1249