Talos Vulnerability Report

TALOS-2020-1133

Microsoft Azure Sphere Capability access control privilege escalation vulnerability

AUGUST 24, 2020

CVE NUMBER

None

SUMMARY

A privilege escalation vulnerability exists in the Capability access control functionality of Microsoft Azure Sphere 20.06. A set of specially crafted ptrace syscalls can be used to obtain elevated capabilities. An attacker can write a shellcode 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-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.

In the Azure Sphere security model an extra layer of protection exists within the traditional Linux capabilities system via custom LSM, a separate Azure Sphere specific credential structure that looks like such:

```
// exposed through /proc/<pid>/attr/exec
struct azure_sphere_task_cred {
    union {
        u8     raw_bytes[16];
        struct azure_sphere_guid guid; // [1]
    } component_id;
    char   daa_tenant_id[64];          // [2]
    bool   is_app_man : 1;             // [3]
    bool   job_control_allowed : 1;
    unsigned int : 0;
    u32 capabilities;                  // [4]
};
```

This `azure_sphere_task_cred` contains the `component_id` of the process that is running [1], and also the `tenant_id` used for attestation [2]. More importantly though are the `is_app_man` field at [3] and capabilities field at [4]. Whenever a new application is run, the `application_manager` (Azure Sphere's `init`) will determine the correct UID to assign the process via the `uid_map` in `/mnt/config`. In examining the `application_manager` binary we also see that these Azure Sphere capabilities are also determined on application start:

```
// application_manager.bin
00022e76  05f23511   addw    r1, r5, #0x135  {component_uid_???}
00022e7a  f3f75bf9   bl      #cmp_component_id
00022e7e  0028       cmp     r0, #0  // d940e448-10b8-4b85-ac5f-69e6a6e4efc6
00022e80  4ed1       bne     #0x22f20

00022e82  baf1090f   cmp     r10, #9
00022e86  18d1       bne     #0x22eba

00022e88  05f24511   addw    r1, r5, #0x145  {component_uid_1002}
00022e8c  3046       mov     r0, r6  // 641f94d9-7600-4c5b-9955-5163cb7f1d75
00022e8e  f3f751f9   bl      #cmp_component_id
00022e92  0028       cmp     r0, #0
00022e94  46d1       bne     #0x22f24

00022e96  05f25511   addw    r1, r5, #0x155  {component_uid_1003}
00022e9a  3046       mov     r0, r6  // 48a22e96-d078-4e34-9d7a-91b3404031da
00022e9c  f3f74af9   bl      #cmp_component_id
00022ea0  0028       cmp     r0, #0
00022ea2  42d1       bne     #0x22f2a

00022ea4  05f26511   addw    r1, r5, #0x165  {component_uid_1001}
00022ea8  3046       mov     r0, r6  // 7ba05ff7-7835-4b26-9eda-29af0c635280
00022eaa  f3f743f9   bl      #cmp_component_id
00022eae  0028       cmp     r0, #0
00022eb0  14bf       ite     ne
```

For each given hardcoded `component_id` in this check there is a corresponding binary in the official firmware that uses it:

```
1001 7ba05ff7-7835-4b26-9eda-29af0c635280 networkd
1002 641f94d9-7600-4c5b-9955-5163cb7f1d75 gatewayd
1003 48a22e96-d078-4e34-9d7a-91b3404031da azured
1004 a65f3686-e50a-4fff-b25d-415c206537af azcore
1005 89ecd022-0bdd-4767-a527-d756dd784a19 rng-tools
```

Continuing on from where the disassembly left off, we can see that the result of the `component_id` checking results in a value being moved into `r0` that corresponds to the previously mentioned `azure_sphere_task_cred->capabilities` field:

```
00022eb2  4ff49070   mov     r0, #0x120 // uid 1001
00022eb6  4ff48070   mov     r0, #0x100 // uid 1004, uid 1005
[...]

00022f20  1020       movs    r0, #0x10  // SFS tenant
00022f22  cae7       b       #0x22eba

00022f24  40f20730   movw    r0, #0x307 // uid 1002
00022f28  c7e7       b       #0x22eba

00022f2a  40f24330   movw    r0, #0x343 // uid 1003
00022f2e  c4e7       b       #0x22eba
```

If we convert a given UID's AZURE_SPHERE capabilities to readable form, we can see for instance that UID 1003 (`azured`) is granted the following capabilities:

```
AZURE_SPHERE_CAP_UPDATE_IMAGE
AZURE_SPHERE_CAP_QUERY_IMAGE_INFO
AZURE_SPHERE_CAP_POSTCODE
AZURE_SPHERE_CAP_RECORD_TELEMETRY
AZURE_SPHERE_CAP_MANAGE_AND_LOG_TELEMTRY
```

It's interesting to note that in order to see ones own processes' Azure sphere capabilities, the `capget()` syscall or `/proc/self/status` will not suffice, instead you must look at `/proc/self/attr/current`:

```
> cat /proc/self/attr/current
CID: AAAAAA-BBBB-CCCC-DDDD-123412341234
TID: 111111-1111-1111-4444-555555555555
CAPS: 00000000
```

This kernel file is defined in `<kernel_root>/security/azure_sphere/lsm.c` in `azure_sphere_security_getprocattr`. More interesting however is that this file also defines a `azure_sphere_security_setprocattr` function, one that permanently changes a given processes Azure Sphere capabilities:

```
static int azure_sphere_security_setprocattr(struct task_struct *p, char *name, void *value, size_t size) {
    struct azure_sphere_security_set_process_details *data = value;
    struct cred *cred; // comp_id[16], bool job_control_allowed, azure_sphere_capability_t capabilities
    struct azure_sphere_task_cred *tsec;
    int ret;

    // Can only set in binary format
    if (strcmp(name, "exec") != 0) {
        return -EINVAL;
    }

    if (value == NULL || size < sizeof(*data)) { // user controlled, max 0x1000
        return -EINVAL;
    }

    if (p != current) {
        // You can only change the current process
        return -EPERM;
    }

    cred = prepare_creds();
    if (!cred) {
        return -ENOMEM;
    }
    tsec = cred->security;    // [1]

    //if no security entry then fail
    if (!tsec) {
        ret = -ENOENT;
        goto error;
    }

    if (!tsec->is_app_man) {  // [2]
        ret = -EPERM;
        goto error;
    }


    memcpy(&tsec->component_id, data->component_id, sizeof(tsec->component_id));
    memset(&tsec->daa_tenant_id, 0, sizeof(tsec->daa_tenant_id));
    memcpy(&tsec->daa_tenant_id, data->daa_tenant_id, strnlen(data->daa_tenant_id, sizeof(tsec->daa_tenant_id) - 1));
    tsec->is_app_man = false; // [3]
    tsec->job_control_allowed = data->job_control_allowed;
    tsec->capabilities = data->capabilities; //[4]

    return commit_creds(cred);
```

The main thing we care about in the above is that by writing to `/proc/self/attr/exec` we get our process' `azure_sphere_task_cred` struct at [1], verify that the `is_app_man` bool is still true [2] and then switch it to false [3], and finally assign the Azure Sphere specific capabilities at [4]. While simple in implementation, the consequences are quite substantial, as the `is_app_man` bool is never referenced anywhere else in the Azure Sphere kernel source code, leaving us with only one intended way to ever gain Azure Sphere capabilities, these capabilities being the sole way of gaining access to Security Manger and Pluton ioctls.

An interesting situation can occur however by utilizing TALOS-2020-1131, TALOS-2020-1132, and TALOS-2020-1137, in which the Azure Sphere device can be manipulated into running our installed application with a UID normally reserved for one of the system UIDs (e.g. 1003, `azured`). In such a situation, we do not have any of the Azure Sphere capabilities reserved for that UID since our application's `component_id` does not match up to any of the `component_id`'s in the `application_manager` disassembly from above.

As discussed in TALOS-2020-1137, we can actually see the `azured` process running via `procfs`:

```
> id
uid=1003 gid=1003 groups=5(gpio)
> ps aux | grep azure
PID   USER     TIME  COMMAND
   30 1003      0:00 /mnt/sys/azured/bin/azured --update --daemonize
```

Intuitively this makes sense, if we're running as the same UID and GID as another process we should be able to see that other process, but let's examine a bit more in depth why this is exactly:

```
//<kernel_root>/fs/proc/root.c

/* for the /proc/ directory itself, after non-process stuff has been done */
int proc_pid_readdir(struct file *file, struct dir_context *ctx)
{
    struct tgid_iter iter;
    struct pid_namespace *ns = file_inode(file)->i_sb->s_fs_info;
    loff_t pos = ctx->pos;

    // [...]

    iter.tgid = pos - TGID_OFFSET;
    iter.task = NULL;
    for (iter = next_tgid(ns, iter); // [1]
         iter.task;
         iter.tgid += 1, iter = next_tgid(ns, iter)) {
        char name[PROC_NUMBUF];
        int len;

        cond_resched();
        if (!has_pid_permissions(ns, iter.task, 2)) //[2]
            continue;

        len = snprintf(name, sizeof(name), "%d", iter.tgid);
        ctx->pos = iter.tgid + TGID_OFFSET;
        if (!proc_fill_cache(file, ctx, name, len,
                    proc_pid_instantiate, iter.task, NULL)) {
            put_task_struct(iter.task);
            return 0;
        }
    }
    ctx->pos = PID_MAX_LIMIT + TGID_OFFSET;
    return 0;
}
```

The above code is run when we do `ls` on `/proc`, as Linux traditionally lists directories corresponding to the different processes currently running. At [1], we see this iteration occurring over each “Thread Group ID”, and if our current process passes the `has_pid_permission()` check, then we get to see that process’ directory. Continuing into `has_pid_permission()`:

```
/*
 * May current process learn task's sched/cmdline info (for hide_pid_min=1)
 * or euid/egid (for hide_pid_min=2)?
 */
static bool has_pid_permissions(struct pid_namespace *pid,
                 struct task_struct *task,
                 int hide_pid_min)
{
    if (pid->hide_pid < hide_pid_min)
        return true;
    if (in_group_p(pid->pid_gid))
        return true;
    return ptrace_may_access(task, PTRACE_MODE_READ_FSCREDS); // [1]
}
```

For our purposes we only care about the line at [1], which implies that if we can see the `/proc` directory for a given process then we have a baseline amount of PTRACE permissions. Continuing into `__ptrace_may_access()`:

```
/* Returns 0 on success, -errno on denial. */
static int __ptrace_may_access(struct task_struct *task, unsigned int mode)
{
    const struct cred *cred = current_cred(), *tcred;
    struct mm_struct *mm;
    kuid_t caller_uid;
    kgid_t caller_gid;
[...]
    /* Don't let security modules deny introspection */
    if (same_thread_group(task, current)) // task->signal == current->signal
        return 0;
    rcu_read_lock();
    if (mode & PTRACE_MODE_FSCREDS) {
        caller_uid = cred->fsuid;
        caller_gid = cred->fsgid;
[...]
    tcred = __task_cred(task);
    if (uid_eq(caller_uid, tcred->euid) &&
        uid_eq(caller_uid, tcred->suid) &&
        uid_eq(caller_uid, tcred->uid)  &&
        gid_eq(caller_gid, tcred->egid) &&
        gid_eq(caller_gid, tcred->sgid) &&
        gid_eq(caller_gid, tcred->gid))
        goto ok;
```

We don’t really have to go too in depth into `__ptrace_may_access`, the only things it seems to care about are that the `resuid()` and `resgid()` of the accessing process matches up with the process being targeted. Since we can already see the `azured` process directory in `/proc` we know by definition that we pass the `__ptrace_may_access()` check, and this fact might lead one to guess where else `__ptrace_may_access()` is used. In the event that astute readers guessed `ptrace()`, then they would be right, and this brings us to the crux of the vulnerability:

```
// <kernel_root>/security/commoncap.c
/**
 * cap_ptrace_access_check - Determine whether the current process may access
 *             another
 * @child: The process to be accessed
 * @mode: The mode of attachment.
 *
 * If we are in the same or an ancestor user_ns and have all the target
 * task's capabilities, then ptrace access is allowed.
 * If we have the ptrace capability to the target user_ns, then ptrace
 * access is allowed.
 * Else denied.
 *
 * Determine whether a process may access another, returning 0 if permission
 * granted, -ve if denied.
 */
int cap_ptrace_access_check(struct task_struct *child, unsigned int mode)
{
    int ret = 0;
    const struct cred *cred, *child_cred;
    const kernel_cap_t *caller_caps;

    rcu_read_lock();
    cred = current_cred();
    child_cred = __task_cred(child);
    if (mode & PTRACE_MODE_FSCREDS)
        caller_caps = &cred->cap_effective;
    else
        caller_caps = &cred->cap_permitted;
    if (cred->user_ns == child_cred->user_ns &&
        cap_issubset(child_cred->cap_permitted, *caller_caps)) // [1]
        goto out;
    if (ns_capable(child_cred->user_ns, CAP_SYS_PTRACE))
        goto out;
    ret = -EPERM;
out:
    rcu_read_unlock();
    return ret;
}
```

With standard Linux capabilities there is an explicit check at [1] to make sure that the attaching process' capabilities are a subset of the target's capabilities, and if this condition is not met, the attaching process must have `CAP_SYS_PTRACE` in order to attach. On the Azure Sphere device we can never get this `CAP_SYS_PTRACE` capability, however there is no explicit check that verifies an attaching process' Azure Sphere capability is a subset of the target's Azure Sphere capabilities. If they had been implemented in the same field of the standard Linux credential structure, this would not be an issue, but as pointed out in the beginning of the writeup, the Azure Sphere specific capabilities are located `cred->security->capabilities`. And this fact allows us to attach to another process with more Azure Sphere capabilities (e.g. any other app running), as long as we share the UID with that process (which can be done by chaining TALOS-2020-1131, TALOS-2020-1132, and TALOS-2020-1137):

```
> /mnt/apps/11223344-1234-1234-1234-aabbccddeeff/bin/busybox id
uid=1001 gid=1001 groups=12(sys-log),500(ipc-appman-reboot),501(ipc-appman-sideload),502(ipc-appman-cloudload),503(ipc-appman-appstart)

> cat /proc/self/attr/current
CID: 48A22E96-D078-4E34-9D7A-91B3404031DA
TID: [...]
CAPS: 00000343
```

Thus in summary: an attacker can use the `ptrace()` API to gain execution in another Azure Sphere process and use its Azure Sphere capabilities to access an entirely new set of IOCTL requests for `/dev/pluton` and all the `/dev/security-monitor` ioctls that likewise could not be utilized before. As a sample of this, in the Security Monitor's SMAPI alone for capability 0x343:

```
// 0x343:
// AZURE_SPHERE_CAP_UPDATE_IMAGE
// AZURE_SPHERE_CAP_QUERY_IMAGE_INFO
// AZURE_SPHERE_CAP_POSTCODE
// AZURE_SPHERE_CAP_RECORD_TELEMETRY
// AZURE_SPHERE_CAP_MANAGE_AND_LOG_TELEMTRY

static azure_sphere_sm_cap_lookup_t azure_sphere_sm_cmd_required_capabilities[] = {

    {.cmd = AZURE_SPHERE_SMAPI_GET_APPLICATION_IMAGE_COUNT, .caps = AZURE_SPHERE_CAP_QUERY_IMAGE_INFO},
    {.cmd = AZURE_SPHERE_SMAPI_LIST_ALL_APPLICATION_IMAGES, .caps = AZURE_SPHERE_CAP_QUERY_IMAGE_INFO},
    {.cmd = AZURE_SPHERE_SMAPI_GET_COMPONENT_IMAGES, .caps = AZURE_SPHERE_CAP_QUERY_IMAGE_INFO},
    {.cmd = AZURE_SPHERE_SMAPI_GET_ABI_TYPE_COUNT, .caps = AZURE_SPHERE_CAP_QUERY_IMAGE_INFO},
    {.cmd = AZURE_SPHERE_SMAPI_GET_ABI_VERSIONS, .caps = AZURE_SPHERE_CAP_QUERY_IMAGE_INFO},
    {.cmd = AZURE_SPHERE_SMAPI_GET_UPDATE_CERT_STORE_IMAGE_INFO, .caps = AZURE_SPHERE_CAP_QUERY_IMAGE_INFO},
    //[...]
    {.cmd = AZURE_SPHERE_SMAPI_SHOULD_IMAGE_BE_UPDATED, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_INVALIDATE_IMAGE, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_OPEN_IMAGE_FOR_STAGING, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_WRITE_BLOCK_TO_STAGE_IMAGE, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_COMMIT_IMAGE_STAGING, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_ABORT_IMAGE_STAGING, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_INSTALL_STAGED_IMAGES, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_GET_COMPONENT_COUNT, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_GET_COMPONENT_SUMMARY, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_STAGE_COMPONENT_MANIFESTS, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_COUNT_OF_MISSING_IMAGES_TO_DOWNLOAD, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_GET_MISSING_IMAGES_TO_DOWNLOAD, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_STAGE_BASE_MANIFESTS, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_COUNT_OF_MISSING_BASE_IMAGES_TO_DOWNLOAD, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_GET_MISSING_BASE_IMAGES_TO_DOWNLOAD, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    {.cmd = AZURE_SPHERE_SMAPI_GET_SOFTWARE_ROLLBACK_INFO, .caps = AZURE_SPHERE_CAP_UPDATE_IMAGE},
    //[...]
    {.cmd = AZURE_SPHERE_SMAPI_PERIPHERAL_ACQUIRE, .caps = AZURE_SPHERE_CAP_PERIPHERAL_PIN_MAPPING},
    {.cmd = AZURE_SPHERE_SMAPI_PERIPHERAL_RELEASE, .caps = AZURE_SPHERE_CAP_PERIPHERAL_PIN_MAPPING},
    {.cmd = AZURE_SPHERE_SMAPI_PERIPHERAL_GET_AVAILABLE_DOMAINS, .caps = AZURE_SPHERE_CAP_PERIPHERAL_PIN_MAPPING},
    {.cmd = AZURE_SPHERE_SMAPI_PERIPHERAL_LOCK_CONFIG, .caps = AZURE_SPHERE_CAP_PERIPHERAL_PIN_MAPPING},
};
```

TIMELINE

2020-07-31 - Vendor Disclosure
2020-08-24 - Public Release

CREDIT

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

VULNERABILITY REPORTS

PREVIOUS REPORT

TALOS-2020-1128

NEXT REPORT

TALOS-2020-1137