Talos Vulnerability Report

TALOS-2020-1117

Microsoft Azure Sphere asynchronous ioctl denial-of-service vulnerability

JULY 31, 2020

CVE NUMBER

CVE-2020-35609

Summary

A denial-of-service vulnerability exists in the asynchronous ioctl functionality of Microsoft Azure Sphere 20.05. A sequence of specially crafted ioctl calls can cause a denial of service. An attacker can write a shellcode to trigger this vulnerability.

Tested Versions

Microsoft Azure Sphere 20.05

Product URLs

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

CVSSv3 Score

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

CWE

CWE-400 - Uncontrolled Resource Consumption

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.

The `/dev/pluton` kernel driver facilitates communication between the normal Linux kernel running on Cortex A7 and the Pluton subsystem on the Cortex M4, which is accessible by any user on the system. It implements very few functions (open, read, poll, close, and ioctl), but provides a decent amount of ioctl requests for interacting with Pluton:

```
#define PLUTON_GET_SECURITY_STATE _IOWR('p', 0x01, struct azure_sphere_get_security_state_result)
#define PLUTON_GENERATE_CLIENT_AUTH_KEY _IOWR('p', 0x06, uint32_t)
#define PLUTON_COMMIT_CLIENT_AUTH_KEY _IOWR('p', 0x07, uint32_t)
#define PLUTON_GET_TENANT_PUBLIC_KEY _IOWR('p', 0x08, struct azure_sphere_ecc256_public_key)
#define PLUTON_PROCESS_ATTESTATION _IOWR('p', 0x09, struct azure_sphere_attestation_command)
#define PLUTON_SIGN_WITH_TENANT_ATTESTATION_KEY _IOWR('p', 0x0A, struct azure_sphere_ecdsa256_signature)
#define PLUTON_SET_POSTCODE _IOWR('p', 0x0B, uint32_t)
#define PLUTON_GET_BOOT_MODE_FLAGS _IOWR('p', 0x0C, struct azure_sphere_boot_mode_flags)
#define PLUTON_IS_CAPABILITY_ENABLED _IOWR('p', 0x0D, struct azure_sphere_is_capability_enabled)
#define PLUTON_GET_ENABLED_CAPABILITIES _IOR('p', 0x0E, struct azure_sphere_get_enabled_capabilities)
#define PLUTON_SET_MANUFACTURING_STATE _IOW('p', 0x0F, struct azure_sphere_manufacturing_state)
#define PLUTON_GET_MANUFACTURING_STATE _IOR('p', 0x10, struct azure_sphere_manufacturing_state)
#define PLUTON_DECODE_CAPABILITIES _IOR('p', 0x11, struct azure_sphere_decode_capabilities_command)
```

Out of these pluton ioctls, we just need to pick one that we have the capabilities for, since most of these ioctls are protected by specific `AZURE_SPHERE_CAP_*` capabilities. While this vulnerability has also been triggered with `PLUTON_DECODE_CAPABILITIES`, for this writeup let's choose `PLUTON_SIGN_WITH_TENANT_ATTESTATION_KEY`:

```
struct azure_sphere_ecdsa256_signature {
    uint8_t R[32];
    uint8_t S[32];
};

#define PLUTON_SIGN_WITH_TENANT_ATTESTATION_KEY _IOWR('p', 0x0A, struct azure_sphere_ecdsa256_signature)

///
/// PLUTON_SIGN_WITH_TENANT_ATTESTATION_KEY message handler
///
/// @arg - ioctl buffer
/// @data - file data for FD
/// @async - is the FD in async mode
/// @returns - 0 for success
int pluton_sign_with_tenant_attestation_key(void __user *arg, struct pluton_file_data *data, bool async) {
    u32 ret = 0;
    struct azure_sphere_task_cred *tsec;
    struct azure_sphere_sign_with_tenant_key request;
    struct azure_sphere_ecdsa256_signature signature;

    // no runtime permission check
    ret = copy_from_user(&request.digest, arg, sizeof(request.digest));
    if (unlikely(ret)) {
        return ret;
    }

    // copy out the tenant id
    tsec = current->cred->security;
    memcpy(&request.tenant_id, tsec->daa_tenant_id, sizeof(request.tenant_id));


    ret = pluton_send_mailbox_message(SIGN_WITH_TENANT_ATTESTATION_KEY,
        &request, sizeof(request), &signature, sizeof(signature), 0, sizeof(signature), data, async);  // [1]

        // no data sent back on err
    if (!ret) {
        ret = copy_to_user(arg, &signature, sizeof(signature));
    }

    return ret;
}
```

Generally all the Pluton ioctls follow a specific pattern like this, so it's not really worth diving too deeply. The main part we actually care about (which is still common to each pluton ioctl) is the `pluton_send_mailbox_message` function at [1]. This function ends up sending a formatted message to the Pluton chip over a shared ring buffer. When Pluton is done processing, it utilizes the same shared buffer to send a message back to the Linux kernel, as one might expect.
An important thing to note though is that this can be done either synchronously or asynchronously, as shown by the `bool async` flag, which is determined by how the user opens up `/dev/pluton`, either with `O_ASYNC` or not. For the purposes of this writeup, we don't really care about synchronous requests, since the vulnerability only triggers with asynchronous requests. Thus, let's follow the asynchronous code path:

```
int pluton_send_mailbox_message(enum pluton_remoteapi_commands cmd,
                            void *data, u32 data_size,
                            void *buffer, u32 buffer_size,
                            size_t data_offset, size_t offset_size,
                            struct pluton_file_data *file_data, bool async){
int ret = 0;

if (async) {
    if (mutex_lock_interruptible(&pending_async_ops_mutex))
        return -ERESTARTSYS;

    if (mutex_lock_interruptible(&file_data->mutex)) {
        ret = -ERESTARTSYS;
        goto unlock;
    }

    if (file_data->data) {
        ret = -EAGAIN;
    } else {
        // malloc space for the message
        file_data->data = kmalloc(buffer_size, GFP_KERNEL);  // [1]
        if (file_data->data == NULL) {
            ret = -ENOMEM;
        } else {
            file_data->data_len = buffer_size;
            file_data->data_offset = data_offset;
            memcpy(file_data->data, buffer, buffer_size);

            // Send async op
            ret = pluton_remote_api_send_command_to_m4_async(  //[2]
                cmd, data, data_size,
                pluton_async_callback, file_data);
```

The main difference between the synchronous and asynchronous pluton ioctls (aside from the synchronicity), is the backing of the data: while the synchronous request eventually does `copy_to_user` back into the ioctl buffer, in the asynchronous request there's an allocation of `sizeof(azure_sphere_ecdsa256_signature)` (0x40 bytes) made [1].
Assuming that the request completes and is successful, the pluton driver will allow this allocation to be read in driver's read function: `pluton_read`. After allocation and setting up the `pluton_file_data` structure correctly, we see the allocated buffer sent at [2]. For a quick insight into the `pluton_file_data`:

```
// Data attached to a /dev/pluton fd
struct pluton_file_data {
    // Wait queue for poll
    wait_queue_head_t waitqueue;
    // Lock for async operations
    struct mutex mutex;
    // Pending data to read
    void *data;
    // Size of pending data
    size_t data_len;
    // Offset to copy data to
    size_t data_offset;
    // Is data ready for read
    bool data_ready;
    // List pointer
    struct list_head list;
};
```

Continuing on, `pluton_remote_api_send_command_to_m4_async` is just a wrapper for `pluton_remote_api_send_command_to_m4`, but we don't really care since that's more ring-buffer based code. Regardless, with all the appropriate context covered, the actual vulnerability: upon quickly sending a continuous stream of asynchronous `/dev/pluton` ioctl requests, a hardware watchdog reset occurs because of an infinite loop in the Pluton ring buffer code. The repeated asynchronous ioctls cause the M4 to A7 ring buffer to fill up completely, and thus when a new empty buffer is sought out by the `/dev/pluton` kernel driver, it never returns and the device reboots, resulting in a denial of service.

Crash Information

By connecting to one of the board's UART, while sending asynchronous ioctl request repeatedly, it's possible to see the device rebooting:

```
[PLUTON] !! ERROR: Key not initialized
[PLUTON] !! ERROR: Key not initialized
[PLUTON] !! ERROR: Key not initialized
[PLUTON] !! ERROR: Key not initialized
[PLUTON] !! ERRO[1BL] BOOT: 70e00000/00000008/01070000
G=...
D=...,N=...
[PLUTON] Logging initialized
[PLUTON] Booting HLOS core
```

Timeline

2020-07-02 - Vendor Disclosure
2020-07-31 - Public Release

CREDIT

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