Talos Vulnerability Report

TALOS-2020-1118

Microsoft Azure Sphere AF_AZSPIO socket memory corruption vulnerability

JULY 31, 2020

CVE NUMBER

CVE-2020-16970

Summary

A memory corruption vulnerability exists in the AF_AZSPIO socket functionality of Microsoft Azure Sphere 20.05. A sequence of socket operations can cause a double-free and out-of-bounds read in the kernel. 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

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

CWE

CWE-415 - Double Free

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 Azure Sphere has a procedure that utilizes special `AF_AZSPIO` sockets to communicate between high-level and real-time applications: https://docs.microsoft.com/en-us/azure-sphere/app-development/high-level-inter-app.
As mentioned, in order to allow this connection to occur in the kernel, one must edit their high-level application's manifest to include: `"AllowedApplicationConnections": [ "005180BC-402F-4CB3-A662-72937DBCDE47" ]`, where by the value is the `component_id` of the high-level app.

But regardless of if this entry is in the application manifest or not, a high-level app can still utilize an `AF_AZSPIO` socket via the normal linux socket api:

```
#define AF_AZSPIO 43

int azspio_sock = socket(AF_AZSPIO, 0x80002, 0x0);
```

Just like any other socket, the creation is pretty simple, it's more the binding and sending addresses that's atypical:

```
#define AZSPIO_COMPONENT_ID_LENGTH 16

struct sockaddr_azspio {
    uint16_t sa_family;           // AF_AZSPIO
    uint16_t sa_port;             // Must be zero.
    uint8_t  sa_component_id[16]; // Must match component_id of app.
};
```

The notes above in the comments must be followed in order for any `bind`, `send`, `recvmsg`, `sendmsg`, to work, and this can be done with the below code:

```
struct sockaddr_azspio sockaddr;
sockaddr.sa_family = AF_AZSPIO;
sockaddr.sa_port = 0x0;

memset(&sockaddr.sa_component_id,0x0,sizeof(sockaddr.sa_component_id));
// make sa_component_id match app's component id.
```

There's also a little complication on converting a `component_id` from string form (`"ComponentId": "b0000000-0000-0000-0000-000000000000"`) to the matching binary form, the first three groups of the `component_id` are in little endian, whereas the last two are big endian. To illustrate, if I have a `uint8_t[16] component_id`, each index in that char array would correspond to the string as such:

```
03 02 01 00 - 05 04 - 07 06 - 08 09 - 0a 0b 0c 0d 0e 0f
```

Thus, in order to get a `component_id` of `b0000000-0000-0000-0000-000000000000`, one must do the following:

```
memset(&sockaddr.sa_component_id,0x0,sizeof(sockaddr.sa_component_id));
sockaddr.sa_component_id[3] = 0xb0;
```

And likewise, then doing `memcpy(&sockaddr.sa_component_id[6],"\xde\xad\xbe\xef",4);` would result in a component id of `b0000000-0000-adde-beef-000000000000`. Moving on, this `component_id` acts essentially as the IP address of the socket (with regards to other apps), and now that we have a valid `sa_component_id`, we can bind our socket:

```
ret = bind(azspio_sock, &sockaddr, sizeof(sockaddr));
```

This bind results in a few different things, but most importantly we hit the following code:

```
// kernel/net/azspio/azspio.c
static int __azspio_bind(struct socket *sock,
             const struct sockaddr_azspio *addr, int zapped) {

    [...]

    if (!azspio_id_matches(component_id, &current_component_id)) {
        return -EACCES;
    }

    /* rebinding ok */
    if (!zapped && !azspio_id_matches(component_id, ipc->us.sa_component_id)) { // [1]
        return 0;
    }

    rc = azspio_socket_assign(ipc);                                              // [2]
    if (rc)
        return rc;

    /* unbind previous, if any */
    if (!zapped) // [3]
        azspio_socket_remove(ipc);

    [...]


    if (!rc) {
        sock_reset_flag(sk, SOCK_ZAPPED);                                        //[4]
    }

    return rc;
}
```

At [1], the `azspio_socket_assign` function adds `ipc->item` into a static linked list (`&azspio_all_sockets`) via the kernel's list api (`list_add`):

```
/* Add socket to global socket list. */
static int azspio_socket_assign(struct azspio_sock *ipc)
{
    mutex_lock(&azspio_socket_lock);
    list_add(&ipc->item, &azspio_all_sockets);
    mutex_unlock(&azspio_socket_lock);
    sock_hold(&ipc->sk);
    return 0;
}
```

Continuing on in `__azspio_bind`:

```
    rc = azspio_socket_assign(ipc);       // [1]
    if (rc)
        return rc;

    /* unbind previous, if any */
    if (!zapped)                          // [2]
        azspio_socket_remove(ipc);

    [...]


    if (!rc) {
        sock_reset_flag(sk, SOCK_ZAPPED); //[3]
    }

    return rc;
}
```

At [2], we see that if the `SOCK_ZAPPED` flag is not set for our socket, the socket immediately gets removed, however since this is a socket that has not been bound before, this is not the case. The only place that an `AF_AZSPIO` socket can have its SOCK_ZAPPED flag unset is at [3], after it has successfully bound to a socket.

Now, something interesting to note about `__azspio_bind` further up:

```
static int __azspio_bind(struct socket *sock,
             const struct sockaddr_azspio *addr, int zapped)
{
    struct azspio_sock *ipc = azspio_sk(sock->sk);
    const u8 *component_id = addr->sa_component_id;
    struct azure_sphere_guid current_component_id;
    struct sock *sk = sock->sk;
    int rc;

    if (!azure_sphere_get_component_id(&current_component_id, current)){          // [1]
        return -EINVAL;
    }

    if (!azspio_id_matches(component_id, &current_component_id)) {                // [2]
        return -EACCES;
    }

    /* rebinding ok */
    if (!zapped && !azspio_id_matches(component_id, ipc->us.sa_component_id)) {   // [3]
        return 0;
    }

    rc = azspio_socket_assign(ipc);                                               // [4]

    if (!zapped)                                                                  // [5]
        azspio_socket_remove(ipc);
```

At [1], [2], and [3], we can see the process of verifying the `component_id` in a couple ways, the most interesting being [3], since if `zapped` is false (i.e. we have already bound the socket), as long as the `component_id` we are binding with matches our application's `component_id`, we continue on. Thus at [4], we can actually add the same socket into the list twice, and since `zapped` is set, we also immediately remove it from this list. Let us now look at such a scenario in memory, first with the declarations and structures in the kernel:

```
struct list_head {
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
```

Thus, `static LIST_HEAD(azspio_all_sockets);` results in:

```
[^~^]> x/2wx &azspio_all_sockets
0xc0620458 <azspio_all_sockets>:        0xc0620458      0xc0620458
```

Moving on, after the first `azspio_socket_assign`:

```
Run till exit from #0  azspio_socket_assign (ipc=<optimized out>) at net/azspio/azspio.c:515
__azspio_bind (sock=<optimized out>, addr=<optimized out>, zapped=1) at net/azspio/azspio.c:562
562     in net/azspio/azspio.c

[o.o]> x/2wx &azspio_all_sockets
0xc0620458 <azspio_all_sockets>:        0xc18d11d8      0xc18d11d8  // [1]

[>_>]> x/2wx 0xc18d11d8
0xc18d11d8:     0xc0620458      0xc0620458                          // [2]
```

We can see both the head [1] and our new socket [2] having `prev` and `next` point to the other as normal. The interesting part happens on the second add:

```
Run till exit from #0  azspio_socket_assign (ipc=<optimized out>) at net/azspio/azspio.c:515
__azspio_bind (sock=<optimized out>, addr=<optimized out>, zapped=0) at net/azspio/azspio.c:562
562     in net/azspio/azspio.c

[~.~]> x/2wx &azspio_all_sockets
0xc0620458 <azspio_all_sockets>:        0xc18d11d8      0xc18d11d8  // [1]

[o.O]> x/2wx 0xc18d11d8
0xc18d11d8:     0xc18d11d8      0xc0620458                          // [2]
```

Our initial list head has not changed, however `sock->item->next` now points to itself [2]. When the first `list_del` happens, immediately after, further corruption happens:

```
Run till exit from #0  azspio_socket_remove (ipc=0xc18d1000) at net/azspio/azspio.c:524
0xc0360786 in __azspio_bind (sock=<optimized out>, addr=<optimized out>, zapped=0) at net/azspio/azspio.c:563
563     in net/azspio/azspio.c

[o.o]> x/2wx &azspio_all_sockets
0xc0620458 <azspio_all_sockets>:        0xc18d11d8      0xc18d11d8

[>_>]> x/2wx 0xc18d11d8
0xc18d11d8:     0x00000100      0x00000200
```

To explain, let us examine `list_del`:

```
# define POISON_POINTER_DELTA 0
/*
 * These are non-NULL pointers that will result in page faults
 * under normal circumstances, used to verify that nobody uses
 * non-initialized list entries.
 */
#define LIST_POISON1  ((void *) 0x100 + POISON_POINTER_DELTA)
#define LIST_POISON2  ((void *) 0x200 + POISON_POINTER_DELTA)

/**
 * list_del - deletes entry from list.
 * @entry: the element to delete from the list.
 * Note: list_empty() on entry does not return true after this, the entry is
 * in an undefined state.
 */
static inline void list_del(struct list_head *entry) {

    __list_del(entry->prev, entry->next);          // [1]
    entry->next = (struct list_head*)LIST_POISON1; // [2]
    entry->prev = (struct list_head*)LIST_POISON2; // [3]
}


static inline void __list_del(struct list_head *prev, struct list_head *next) {

    next->prev = prev;
    prev->next = next;
}
```

Due to the corrupted list, the `__list_del` at [1] actually does nothing, and then our socket's list members get poisoned with 0x200 and 0x100. This leaves us in an interesting position since, if `azspio_socket_remove` gets called again, we will end up dereferencing 0x100 and 0x200.
Assuming we close our socket now and destroy our last reference to the socket file descriptor, `azspio_release` gets called, which will cause another `azspio_socket_remove` to happen, resulting in a crash, and it's worth noting that this also occurs if the process exits. Assuming that one could map the NULL page, this would more than likely result in a privilege escalation, but until a secondary vulnerability is found to map the NULL page, this vulnerability would only cause a denial-of-service.

Crash Information

```
[    7.763014] Unable to handle kernel NULL pointer dereference at virtual address 00000104
[    7.770197] 2de10000-2de33000 r-xp 00000000 fe:00 99 libc++runtime.so.1
[    7.771283] 2de33000-2de34000 r--p 00013000 fe:00 99 libc++runtime.so.1
[    7.771735] 2de34000-2de35000 rw-p 00014000 fe:00 99 libc++runtime.so.1
[    7.771967] 2de35000-2de5d000 r-xp 00000000 fe:00 62 libgcc_s.so.1
[    7.772141] 2de5d000-2de5e000 rw-p 00018000 fe:00 62 libgcc_s.so.1
[    7.772470] 2de5e000-2de7b000 r-xp 00000000 fe:00 96 libapplibs.so.0
[    7.772770] 2de7b000-2de7c000 rw-p 0000d000 fe:00 96 libapplibs.so.0
[    7.773654] 2de7c000-2ded1000 r-xp 00000000 fe:00 61 ld-musl-armhf.so.1
[    7.774182] 2dee0000-2dee2000 rw-p 00054000 fe:00 61 ld-musl-armhf.so.1
[    7.775085] 2dee2000-2dee3000 rw-p 00000000 00:00 0
[    7.775388] 2deed000-2deee000 r-xp 00000000 fe:00 19 azspio_rebind_crash_second
[    7.775900] 2defd000-2defe000 r--p 00000000 fe:00 19 azspio_rebind_crash_second
[    7.776243] 2defe000-2deff000 rw-p 00001000 fe:00 19 azspio_rebind_crash_second
[    7.776694] bed6d000-bed6e000 r-xp 00000000 00:00 0 [sigpage]
[    7.776966] bed70000-bed91000 rw-p 00000000 00:00 0
[    7.777510] pgd = c19e0000
[    7.777835] [00000104] *pgd=00000000
[    7.779802] Internal error: Oops: 805 [#1] THUMB2
[    7.780931] CPU: 0 PID: 50 Comm: azspio_rebind_c Not tainted 4.9.213-mt3620-azure-sphere+ #55
[    7.781678] Hardware name: Generic DT based system
[    7.782294] task: c1a7ed00 task.stack: c0d74000
[    7.783440] PC is at azspio_socket_remove+0x38/0x50
[    7.783826] LR is at mutex_lock+0x9/0x30
[    7.784248] pc : [<c03604a0>]    lr : [<c0369259>]    psr: 60000033
[    7.784248] sp : c0d74f28  ip : 00000000  fp : c0d78608
[    7.784791] r10: 00000008  r9 : c1444020  r8 : 00000000
[    7.784959] r7 : c1444020  r6 : 00000000  r5 : 00000100  r4 : c1942000
[    7.785854] r3 : c19423d8  r2 : 00000100  r1 : 00000200  r0 : c062044c
[    7.786276] Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA Thumb  Segment user
[    7.786659] Control: 50c5387d  Table: 419e0059  DAC: 00000055
[    7.786924] Process azspio_rebind_c (pid: 50, stack limit = 0xc0d74210)
[    7.787188] Stack: (0xc0d74f28 to 0xc0d75000)
[    7.788091] 4f20:                   00800000 c1942000 c1444000 c0360afb c1444000 c185d010
[    7.788714] 4f40: c14432a8 c02cc819 c0d78600 c02cc86d c02cc865 c0193d91 00000000 00000000
[    7.789108] 4f60: c0d620c0 c1a7f024 c0d78600 c1a7ed00 00000000 c0622194 c0d74000 00000006
[    7.789430] 4f80: 00000000 c01240ab c0d74000 c0105ce4 c0d74fb0 00000006 c0105ce4 c0107eb5
[    7.789952] 4fa0: 00000000 00000000 00000000 c0105b61 00000000 00000000 00000000 00000000
[    7.790347] 4fc0: 00000000 00000000 00000000 00000006 00000001 00000000 00000000 00000000
[    7.790787] 4fe0: bed80df0 bed80ddc 2debf225 2debee58 60000030 00000003 00000000 00000000
[    7.792493] [<c03604a0>] (azspio_socket_remove) from [<c0360afb>] (azspio_release+0xb7/0xd0)
[    7.792944] [<c0360afb>] (azspio_release) from [<c02cc819>] (sock_release+0xd/0x58)
[    7.793153] [<c02cc819>] (sock_release) from [<c02cc86d>] (sock_close+0x9/0xc)
[    7.793710] [<c02cc86d>] (sock_close) from [<c0193d91>] (__fput+0x65/0x140)
[    7.793923] [<c0193d91>] (__fput) from [<c01240ab>] (task_work_run+0x4f/0x78)
[    7.794229] [<c01240ab>] (task_work_run) from [<c0107eb5>] (do_work_pending+0x71/0x74)
[    7.794500] [<c0107eb5>] (do_work_pending) from [<c0105b61>] (slow_work_pending+0x9/0x18)
[    7.795003] Code: f44f 7100 f44f 7580 (6053) 601a
[    7.796026] ---[ end trace 78a837ddc818fa2e ]---
[    7.796440] Kernel panic - not syncing: Fatal exception
[    8.805127] Reboot failed -- System halted
```

Timeline

2020-07-02 - Vendor Disclosure

2020-07-31 - Public Release

CREDIT

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