Talos Vulnerability Report

TALOS-2020-1093

Microsoft Azure Sphere Normal World application /proc/self/mem unsigned code execution vulnerability

JULY 31, 2020

CVE NUMBER

CVE-2020-16994

Summary

A code execution vulnerability exists in the normal world's signed code execution functionality of Microsoft Azure Sphere 20.05. A specially crafted shellcode can cause a process' non-writable memory to be written. An attacker can execute a shellcode that modifies the program at runtime via /proc/self/mem to trigger this vulnerability.

Tested Versions

Microsoft Azure Sphere 20.05

Product URLs

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

CVSSv3 Score

6.2 - CVSS:3.0/AV:L/AC:L/PR:N/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 `/proc/pid/mem`.
In 2011, the commit “198214a7ee50375fa71a65e518341980cfd4b2f0” in the Linux source enables writing of `/proc/pid/mem` by default:

```
diff --git a/fs/proc/base.c b/fs/proc/base.c
index e34c3c33b2de..bc15df390ec4 100644
--- a/fs/proc/base.c
+++ b/fs/proc/base.c
@@ -854,10 +854,6 @@ out_no_task:
         return ret;
 }

-#define mem_write NULL
-
-#ifndef mem_write
-/* This is a security hazard */
 static ssize_t mem_write(struct file * file, const char __user *buf,
                          size_t count, loff_t *ppos)
 {
@@ -914,7 +910,6 @@ out_task:
 out_no_task:
         return copied;
 }
-#endif
```

This code has not been restricted in the Azure Sphere Linux kernel, and can thus be abused by a running process to modify itself: it’s enough to write to `/proc/self/mem` in order to modify the `.text` section of the running program, even though it’s mapped to a page without write permissions.

In pseudo-code this would be as such:

```
int fd = open("/proc/self/mem", O_WRONLY);
lseek(fd, func, 0);                         // alignment
write(fd, shellcode, shellcode_len);        // overwrite the function "func"
```

This sequence of commands overwrites the function pointed by `func` with an arbitrary shellcode, and could be used by an attacker to run unsigned code, after compromising an application.
Finally note that since the scope of this issue is within an already compromised application, the pseudo-code above would have to be implemented via ROP gadgets.

Timeline

2020-06-05 - Vendor Disclosure
2020-07-31 - Public Release
2020-11-10 - CVE assigned

CREDIT

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