Talos Vulnerability Report

TALOS-2020-1090

Microsoft Azure Sphere Normal World application ptrace unsigned code execution vulnerability

JULY 31, 2020

CVE NUMBER

CVE-2020-16991

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 uses the ptrace system call 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 writeable cannot be marked as executable, likewise memory that has been marked executable cannot be marked as writable. 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?". Therefore, while the following code snippets will be in C, in a real situation the code snippets would be the equivalent code in ROP gadgets. A pseudo-code summary of the

simple PoC code flow is as such:

```
int * pc;
__asm__("mov %0, pc" : "=r"(pc));   // [1]

cpid = fork();                      // [2]
if(cpid == 0) {
    while(1) {
        sleep(1);
        send(client_sock,"[^_^] boop ",10, 0);
    }
    send(client_sock,"[>_<] This should never happen\n", 31, 0);   // [3]
}
elif (cpid > 0) {                   // [4]
    unsigned int nop_dubz = 0xbf00bf00; // nop; nop;  in thumb
    int * dst_addr = pc+0x6a; // can be any offset

    ret = ptrace(PTRACE_ATTACH, child, NULL, NULL);               // [5]
    ptrace(PTRACE_POKETEXT , cpid, dst_addr, nop_dubz);           // [6]
    ptrace(PTRACE_CONT, child, NULL, NULL);
}
```

For those unfamiliar with `ptrace`, it's the underlying syscall used by debuggers in order to attach to other processes and do anything that a debugger might want to do, and in our case, edit process memory.

To start, the program saves its `$pc` register [1] for later use and then forks at [2].
The child process immediately enters a loop and periodically sends a heartbeat over a socket (just for testing purposes).
At [3], we note the code path that should never be hit, as the while loop before it should never end.
The parent simultaneously hits the code path at [4], and does some simple math to see where it wants to eventually write to. Due to the nature of `fork()`, the process memory layout is the same between child and parent process regardless of ASLR, so this simple math suffices.
At [5], we attach to the child process via `PTRACE_ATTACH`, which allows us to now write to the child's memory space at [6], where we write two `nops` to the end of the child's infinite loop (replacing the unconditional loop branch).
The `PTRACE_CONT` [7] then resumes the child's execution, the output of which looks like so:

```
~/# ncat 192.168.35.2 61166
[^_^] boop
[^_^] boop
[>_<] This should never happen
```

This shows that it's possible to modify a running program (whose code is in a page with `r-x` permissions). Thus an attacker, after compromising an application, would be able to exploit the `ptrace` syscall to run unsigned code.

Because we have not yet reversed fully the page permission protection functionality yet, we cannot say for certain why this occurs. Our best guess is that the hardware memory protection only monitors for page permission changes and not for direct writes into non-writable pages like `PTRACE_POKE` is capable of. For an indepth discussion about ptrace with COW page behavior, see https://yarchive.net/comp/linux/ptrace_mmap.html.

Timeline

2020-06-03 - Vendor Disclosure
2020-07-28 - Public Release

CREDIT

Discovered by Lilith >_> and Claudio Bozzato of Cisco Talos
