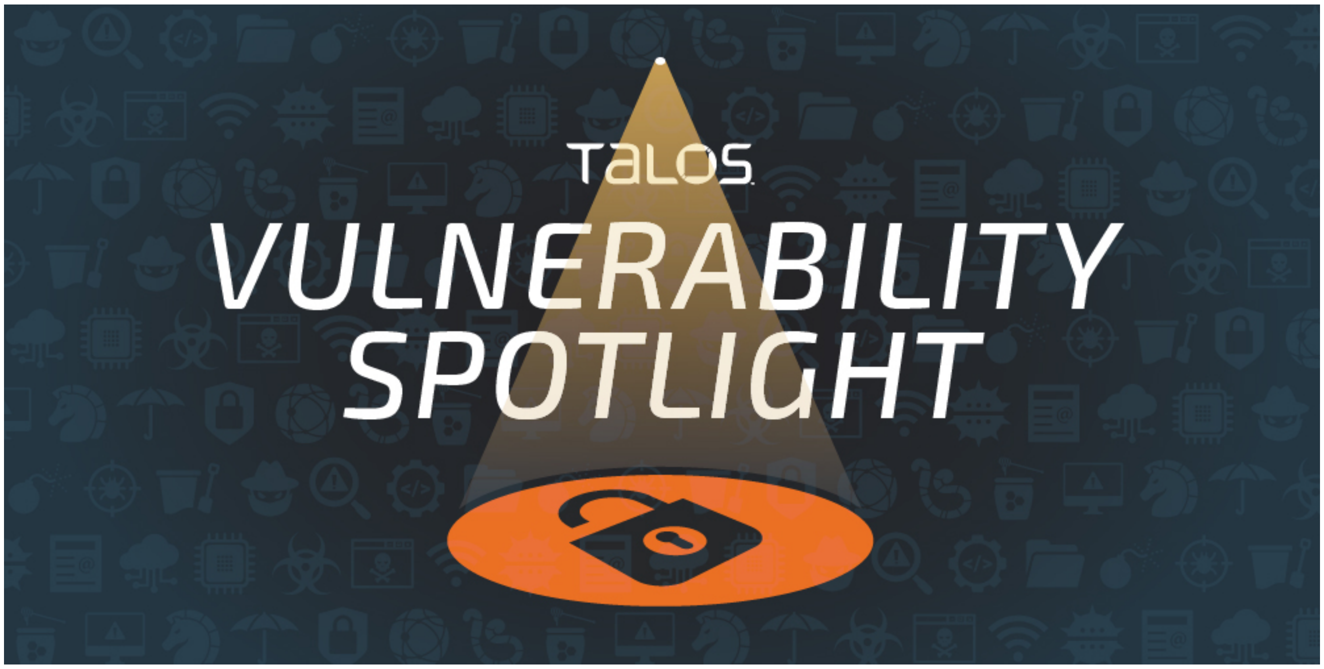
Cisco Talos finds 10 vulnerabilities in Azure Sphere's Linux kernel, Security Monitor and Pluton

blog.talosintelligence.com/2021/11/cisco-talos-finds-10-vulnerabilities-in.html



By Claudio Bozzato and Lilith [-_-];.

Following our previous engagements (see blog posts 1, 2, 3 and 4) with Microsoft's Azure Sphere IoT platform, we decided to take another look at the device, without all the rush and commotion that normally entails a hacking challenge.

Today, we're disclosing another 10 vulnerabilities in Azure Sphere — two of which are on the Linux side, seven that exist in Security Monitor and one in the Pluton security subsystem.

As opposed to our previous architectural overview, this post will simply walk through the vulnerabilities we discovered as part of our continued research into Azure Sphere, starting with the Linux kernel side.

Kernel information disclosure

Microsoft Azure Sphere Kernel GPIO_SET_PIN_CONFIG_IOCTL information disclosure vulnerability (TALOS-2021-1339/CVE-2021-41374)

Azure Sphere allows applications to manage the set of GPIO pins declared in their manifest by sending an *ioctl* to */dev/gpiochip0*. This vulnerability allows an unprivileged attacker using the *GPIO_SET_PIN_CONFIG_IOCTL* to specify an arbitrary *lineoffsets* field and trigger an out-of-bounds read from a kernel structure. This, in turn, partially leaks kernel memory.

Microsoft Azure Sphere Kernel GPIO_GET_PIN_ACCESS_CONTROL_USER information disclosure vulnerability (TALOS-2021-1340/CVE-2021-41375)

Similar to the issue above, this one affects the *GPIO_GET_PIN_ACCESS_CONTROL_USER ioctl* but it requires an attacker to have the *CAP_SYS_ADMIN* capability. Note that even though, normally, having *CAP_SYS_ADMIN* is equivalent to having root privileges, this does not hold true in Azure Sphere. The issue is unsigned to signed integer conversion, which again affects the *lineoffsets* field, and can be exploited to arbitrarily leak kernel memory.

Pluton denial-of-service

Microsoft Azure Sphere Pluton concurrent syscalls denial-of-service vulnerability (TALOS-2021-1347)

This vulnerability shows that by making concurrent Pluton *ioctls*, an unprivileged application can hit a rate limit in Pluton, which is handled with a device reboot. This has not been fixed by Microsoft, since it’s considered an intended behavior.

An application normally requires a capability to reboot this device, this vulnerability allows an application to bypass that requirement, however, Microsoft does not recognize these kinds of privilege escalations as vulnerabilities, so they elected not to assign a CVE.

Security Monitor denial-of-service

Microsoft Azure Sphere Security Monitor SECTION_ABIDepends denial-of-service vulnerability (TALOS-2021-1311)

Every firmware image on the device has a metadata section, which contains multiple sub-sections with various information. Currently, we have documented the following metadata sub-sections, those marked with a “*” are the ones parsed somewhere in Pluton or Secmon:

```
* Debug = 0x4244

  LegacyABIDepends = 0x4441

  Identity  = 0x4449

* ABIDepends = 0x444E

  Legacy = 0x474C

* Signature = 0x4753

* Compression = 0x4D43

  RequiredFlashOffset = 0x4F52

  LegacyABIProvides = 0x5041

* ABIProvides = 0x504E

* TemporaryImage = 0x5054

  Revocation = 0x5652
```

Each of these subsections has its own structure, but first, a hexdump to demonstrate the metadata section:

```
struct metadata_footer {

    uint32_t magic_header;

    uint32_t num_subsections;

    struct metadata_section[num_subsections];

}

struct metatdata_section {

uint16_t tag;

uint16_t size;

<start of per-type data>

}
```

For the current vulnerability, we examine the ABIDepends subsection, whose data is essentially:

```
struct ABIDepends {

          uint32_t size;

          uint64_t ABIdata[size/8];

   }
```

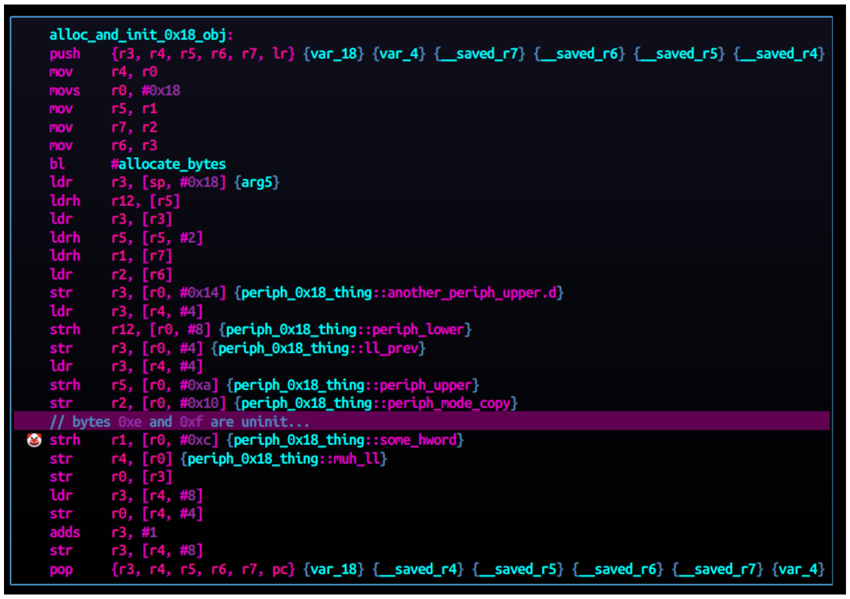
This is a fairly straightforward bug — the *size* field inside the *ABIDepends* structure is just never checked, while it's also the sole terminator for a reading loop. If we set it high enough within an application that we flash, we easily hit an out-of-bounds read into unmapped memory, causing the device to reboot. Since the image is flashed (i.e. persistent) on boot, the image processing functionality triggers immediately and we get a persistent boot loop (until manual recovery).

Security Monitor “post-kernel” bugs

The last set of vulnerabilities could only be reached after gaining full privileges in the Linux Normal World. Lilith Wyatt will be discussing how this exploit works in detail at Hitcon 2021 on Nov. 26, be sure to check out her talk. We’ll also release a detailed blog post about it that day.

Microsoft Azure Sphere Security Monitor SMSyscallPeripheralAcquire information disclosure vulnerability (TALOS-2021-1309)

Starting with *SMSyscallPeripheralAcquire*, the syscall used for configuring the pin muxing to different devices — it contains an information leak in the output buffer that stems from a missing initialization in the object that is populated with output data.



```
alloc_and_init_0x18_obj:
push    {r3, r4, r5, r6, r7, lr} {var_18} {var_4} {__saved_r7} {__saved_r6} {__saved_r5} {__saved_r4}
mov     r4, r0
movs    r0, #0x18
mov     r5, r1
mov     r7, r2
mov     r6, r3
bl      #allocate_bytes
ldr     r3, [sp, #0x18] {arg5}
ldrh    r12, [r5]
ldr     r3, [r3]
ldrh    r5, [r5, #2]
ldrh    r1, [r7]
ldr     r2, [r6]
str     r3, [r0, #0x14] {periph_0x18_thing::another_periph_upper.d}
ldr     r3, [r4, #4]
strh    r12, [r0, #8] {periph_0x18_thing::periph_lower}
str     r3, [r0, #4] {periph_0x18_thing::ll_prev}
ldr     r3, [r4, #4]
strh    r5, [r0, #0xa] {periph_0x18_thing::periph_upper}
str     r2, [r0, #0x10] {periph_0x18_thing::periph_mode_copy}
// bytes 0xe and 0xf are uninit...
strh    r1, [r0, #0xc] {periph_0x18_thing::some_hword}
str     r4, [r0] {periph_0x18_thing::muh_ll}
str     r0, [r3]
ldr     r3, [r4, #8]
str     r0, [r4, #4]
adds    r3, #1
str     r3, [r4, #8]
pop     {r3, r4, r5, r6, r7, pc} {var_18} {__saved_r4} {__saved_r5} {__saved_r6} {__saved_r7} {var_4}
```

Depending on the number of mux pins that are being configured, multiple instances of this structure are created and pieces are directly copied into the output buffer, including a four-byte copy of offset 0xc of the above structure. Since bytes 0xe and 0xf are uninitialized, we get a two-byte secmon heap leak for each pin that's configured, assuming the output buffer has 0x18 bytes for each configured pin.

While not the most informative info leak, there is also a more dangerous one.

Microsoft Azure Sphere Security Monitor SMSyscallWriteBlockToStageImage information disclosure vulnerability (TALOS-2021-1310)

This vulnerability lies inside the *SMSyscallWriteBlockToStageImage* function. The function prototype is:

> int32_t SMSyscallWriteBlockToStageImage(int64_t* handle_value, size_t offset_into_staging_partition, size_t offset_into_srcbuffer, void * srcptr, size_t srcbuffer_size)

The first parameter, the *handle_value*, is essentially a file descriptor that's returned by *SMSyscallOpenImageForStaging*, whose prototype is as follows:

> int32_t SMSyscallOpenImageForStaging(int32_t image_size, int32_t clobbered, uint64_t* output_handle)

An *image_size* is passed in, and if there's enough staging memory available, an *output_handle* is used for the further image staging syscalls. Backing up to *SMSyscallWriteBlockToStageImage*, the *offset_into_staging_partition* variable is how far we're flashing into the current staging image, and this value is checked against the *image_size* from *SMSyscallOpenImageForStaging*.

The fourth and fifth arguments, *srcptr* and *srcbuffer_size* are exactly what one would expect, the source of the copy. The third argument, *offset_into_srcbuffer* is an extreme outlier here, as you can only write ~0x1020 bytes in the first place; there's a defacto hard-limit in place that's enforced when copying from DMA memory to Secmon's private buffer: the total size of the syscall struct and the buffers inside cannot exceed 0x1060.

Why have an *offset_into_srcbuffer* if this offset is on a buffer with a maximum size in the first place? Regardless of the rationale, this value is not checked, which results in us “staging” an image to flash with any data from Secmon's memory space. We can then read this data out via the *SMSyscallReadFlash* syscall. Effectively, this allows us to read data from the entire Secmon memory space.

Microsoft Azure Sphere Security Monitor SMSyscallStageBaseManifests offset calculation out-of-bounds read vulnerability (TALOS-2021-1343/CVE-2021-41376)

The *SMSyscallStageBaseManifests* syscall can be used to stage base manifests. Even though a manifest is wrapped in a standard Azure Sphere image, its contents are undocumented. Inside all of the base manifests that we’ve found, we can find the image and component ID for the Trusted Keystore and the update-cert-store. Only after this manifest is staged, the images defined therein can be flashed. This syscall function prototype is the following:

> int32_t SMSyscallStageBaseManifests(uint32_t offset, char *src_buffer, uint32_t manifest_length)

The *manifest_length* tells the length of the manifest which is expected to be found at some *offset* after *src_buffer*. Similarly to the other Secmon syscalls, the *manifest_length* can’t be larger than 0x1060. The *offset* parameter however has no constraints and it is never checked before or during the syscall execution, and it could be used by an attacker to reference a manifest out-of-bounds. The exploitation however is not trivial, since there are several constraints at play in the manifest headers: in the advisory we show how we can use this out-of-bounds read to stage the manifest in DMA memory, so that it can be read by a secondary M4 core and use a TOCTTOU to alter the manifest while it’s getting staged by this syscall, possibly leading to an information leak.

Microsoft Azure Sphere Security Monitor SMSyscallStageBaseManifests image validation signature check bypass vulnerability (TALOS-2021-1342/CVE-2021-42300)

The *SMSyscallStageBaseManifests* syscall is supposed to verify base manifests (normally signed by Microsoft) before staging them. Manifests are bundled in an Azure Sphere image, which is the same file type shared with firmware images, applications and the keystore, among other functions. One of the header fields of Azure images is the image type, which, in the case of base manifests, should hold a specific value. This advisory shows that by changing the image type of an image manifest, an attacker could bypass the signature check-in *SMSyscallStageBaseManifests*, and stage arbitrary manifests. This, in turn, allows anyone to flash any Microsoft-signed binary, meaning they could downgrade arbitrary firmware (or the whole device) to a previous version, and target older issues in the code.

Microsoft Azure Sphere Security Monitor SMSyscallCommitImageStaging stage-without-manifest denial of service vulnerability (TALOS-2021-1341)

The Trusted Keystore is used for image verification at boot time and when installing images. It’s possible to flash any Microsoft-signed Trusted Keystore without the need to flash a related base manifest. This can be used to flash an old Trusted Keystore from version 20.01 that prevents any further verification of the installed firmware images, preventing the device from booting. This is a denial-of-service that requires manual recovery.

Note: This vulnerability is theoretical and was discovered in development mode but has not been confirmed in pre-production or production environments by either Talos or Microsoft. See the advisory for more details.

Microsoft Azure Sphere Security Monitor SMSyscallCommitImageStaging 1BL firmware downgrade vulnerability (TALOS-2021-1344)

The 1BL is the bootloader used by Azure Sphere. A new version is usually flashed on every firmware upgrade. This vulnerability shows that it is possible to install any 1BL version (thus it’s possible to downgrade its version) by staging the corresponding recovery manifest, which is found in every firmware release. This would allow an attacker to downgrade the 1BL and target older issues in the code.

Note: This vulnerability is theoretical and was discovered in development mode but has not been confirmed in pre-production or production environments by either Talos or Microsoft. See the advisory for more details.

Microsoft issued the following statement on the two vulnerabilities listed above:

> *We thank Cisco Talos for sharing their continued research into Azure Sphere, which first started during the Azure Sphere Security Research Challenge in 2020. After reviewing the findings on TALOS-2021-1341 and TALOS-2021-1344, Microsoft believes the approach described is implemented by design and does not present a security risk to customer production environments. For more information on our conclusion, please read our full statement on these Cisco Talos advisories.*

Coverage

The following SNORTⓇ rules will detect exploitation attempts. Note that additional rules may be released at a future date and current rules are subject to change pending additional vulnerability information. For the most current rule information, please refer to your Cisco Secure Firewall or Snort.org.

Snort Rules: 57745-57746, 57266-57267, 57747-57748, 57888-57889, 57899-57900, 57934-57935, 57963-57964.