Security Now! #955 - 01-02-24

The Mystery of CVE-2023-38606

This week on Security Now!

After everyone is updated with the state of my still-continuing work on SpinRite 6.1, and after I've shared a bit of feedback from our listeners, the entire balance of this first podcast of 2024 will be invested in the close and careful examination of the technical details surrounding something that has never before been found in Apple's custom proprietary silicon.

As we will all see and understand by the time we're finished here today, it is something that can only be characterized as a deliberately designed, implemented and protected backdoor that was intended to be, and was, let loose and present in the wild.

After we all understand what Apple has done through five successive generations of their silicon, today's podcast ends, as it must, by posing a single one-word question: **Why?**

Please use the term "Ad Hoc" in a sentence:

"This is a photo of an ad hoc EU to US power adapter"

SpinRite

As I mentioned two weeks ago, I was hoping to be able to announce that SpinRite 6.1 was finished. And it really is all but finished. If I were not still making progress that matters, I would declare it finished. I've been wrestling with the behavior of some really troubled drives, and I think we're there, but the paint is still wet. So I need to give it a little more time to settle. What I know is that as a consequence of this intense scrutiny at the finish line, this SpinRite does more — much more — than any version of SpinRite ever has before. And these final touches will be inherited by SpinRite 7, so it's not time wasted.

Closing the Loop

Remy van Elst

Location: Rotterdam, The Netherlands
Subject: Which root certificates should you trust? I made the tool requested in 951

Hi Steve,

Happy Spinrite customer here. Can't wait for 6.1.

In episode #951 you discussed root certificates and Leo asked for a "Prune your CA" app. You said: "The good news is this is generic enough that somebody will do it by the next podcast."

Well, fun challenge and I happen to know a bit of C++, so here is my attempt:
https://raymii.org/s/software/Which_Root_Certificates_Should_You_Trust_CertInfo.html

*It parses the Chrome or Firefox browser history and fetches all certificates. It's a reporting-only tool that **does not prune your root store**, since I suspect that could break stuff. So if people want that, they can figure out how to do that themselves (and make a backup).*

I've compiled it as a 32-bit Qt 5.15 C++ application, specially for you, so it runs on Windows 7 and up. Qt 6 (the current version of the C++ framework) does not support Windows 7.

Cheers, Remy

Since I found this as I was pulling together today's podcast I have not yet had the chance to examine it closely. But I quickly went over to Remy's page and it looks like he has done a very nice piece of work. He provides an installer for Windows users and the full source code. So for Linux users he suggests installing QT and compiling his "CertUtil" from the provided source code. He has deliberately removed all Google ads, Google and other tracking from his site, so he suggests that if you find his work useful, you might perhaps consider buying him a cup of coffee.

The link to Remy's page is this week's GRC shortcut, so it's http://grc.sc/955. And Remy... thanks for this. Once I have the chance I'll definitely fire it up and see what it shows!

Martin Bergek / @martinbergek
Referring to last week's "Schrödinger's Bowls" picture of the week:

> *Showed the picture of the week to my son (13 yrs). He immediately gave the obvious answer: "Break the top glass - obviously that is cheaper to replace than the plates."*
> *Slightly annoyed I didn't come up with it myself but happy that the future is looking bright.*

Very clever and very nice, Martin! ... and I completely agree that breaking the empty upper shelf's glass and rescuing the bowls from their precarious plight is a wonderful answer!

@sammysammy222

> *@sggrc Steve, I planned to use DNS Benchmark today. When I ran the download by VirusTotal, 3 security vendors flagged this file as malicious. While I trust you, I wanted to check-in in case there is a problem with the file. Thanks!*

I just dropped the file on VirusTotal and at this moment 4 of the A/V tools dislike it. I'll note, however, that they're all rather obscure A/V. And since GRC's DNS Benchmark is approaching 8.8 million downloads at a rate of 4,000 per day, I suppose it's a good thing that not everyone is as cautious as "sammysammy222" or there would be a lot more concern. But seriously, I poked around a bit and looked at the behavior that concerns VirusTotal and I can't say that I blame it. The program's behavior, if you didn't know better, would peg it as some sort of bizarre DNS DDoS zombie. It's scan network adapters, is chock full of DNS domain names, and has all sorts of mysterious networking code.

Unfortunately, this is the world we're living in today. It doesn't matter that it's never hurt a living soul and never would. It's even triple-SHA256 signed, with various of GRC's old and new certificates — all valid. Does that matter? Apparently not. It still looks terrifying.

But to answer your question SammySammy: The file's code hasn't changed in nearly five years and we've never had anyone raise any actual alarm. So, yeah... I'd say that you can rely upon it.

Christian Nilsson / @chindotse

> *Hello Steve, I'm one of all who is so glad you decided to go past 3E7 hex. Thank you. Speaking of Tailscale etc. I strongly recommend Nebula Mesh. Open source all the way and very easy to manage. I know it has been on your radar before. What is the reason for not advocating it in SN 953?*

No reason at all. I just forgot to mention it among all of the others. And I agree 100% that Nebula Mesh, which was created by guys at Slack, is a very nice looking overlay network. They wrote it for their own use because nothing else that existed at the time, this was back in 2019, did what they needed out of the box. Nebula Mesh's home page on GitHub is under SlackHQ https://github.com/slackhq/nebula and I have the link in the show notes, but you can just search for "nebula mesh" to locate it. It's a fully mutually authenticating peer-to-peer network that can scale from a few to tens of thousands of endpoints. It's portable, written in GO, and has clients for Linux, MacOS, Windows, iOS, Android and even my favorite FreeBSD Unix.

Ethan Stone / @ethstone

Hi Steve. I've noticed something recently that you might be interested in. First on my Windows 10 machine, then on my Windows 11 machine, the Edge browser has kept itself running in the background after I close it. The only way to actually shut it down is to go into Task Manager and manually stop the many processes that keep running. I noticed this because, as one of your most paranoid listeners (I know it's a high bar), I have my browsers set to delete all history and cookies when they shut down and I have CC Cleaner doing the same thing if they don't. CC Cleaner seems to have caught on to Microsoft's little scheme and now notes that Edge is refusing to shut down and asking me if I want to force it. Anyway, it seems like it's just another little scheme to keep all of my data and activity accessible to their grubby little data selling schemes.

I love the show and I'm looking forward to new ones in 2024 and the advent of Spinrite 6.1, although I really, really, really need native USB support. So I hope Spinrite 7 isn't far behind. Also, I'm looking forward to not having to login to Twitter to send you messages, although please give priority to Spinrite 7.

I've been gratified with the feedback, from Twitter users, that they're looking forward to abandoning Twitter to send these sorts of feedback tidbits. My plan is to first get SpinRite 6.1 finished. Then I need an eMail system, which I don't currently have, in order to notify all v6.0 owners of the availability of what has turned out to be a massive free upgrade to 6.0. So that need will create GRC's eMail system and we'll experiment with my idea for preventing all incoming SPAM without using any sort of heuristic false-positive tending filter. And then I plan to immediately begin working on SpinRite 7. I don't want to wait for several reasons. For one thing, I currently **AM** SpinRite. I have the entire thing in my head right now. My dreaming at night is about SpinRite. And I know from prior experience that I will lose it if I don't use it. So I want to immediately dump what's in my head into the next SpinRite code. Another reason is what Ethan referred to about USB. Unfortunately, USB is still SpinRite 6.1's Achilles heel. All of SpinRite has historically run through the BIOS, but I've become so spoiled now by 6.1's direct access to drive hardware that the BIOS now feels like a real compromise. Yet even under 6.1, all USB devices are still being accessed through the BIOS. Since SpinRite's next platform, RTOS-32, already has USB support – though not the level of support required for data recovery – it made no sense for me to spend more time developing USB drivers from scratch for DOS when at least something I can start with is waiting for me under RTOS-32. So, yes, I'm a hurry to get going on SpinRite 7.

And Ethan replied, writing:

OK, but please spend less time on the email thing than you did on SQRL ;)

And, yes, I am not rolling my own from scratch this time. I've already purchased a very nice PHP-based system that I've just been waiting to deploy. I'll be making some customizations to part of it, but it already largely does everything I need!

The Mystery of CVE-2023-38606

Our long-time listeners may recall that during the last year or so we mentioned an infamous and long-running campaign known as "Operation Triangulation" which is an attack against the security of Apple's iOS-based products. This breed of malware gets its name from the fact that it employs canvas fingerprinting to obtain clues about the user's operation environment. It uses WebGL, the Web Graphics Library to draw a specific yellow triangle against a pink background. It then inspects the exact high-depth colors of specific pixels that were drawn because it turns out that different rounding errors used in differing environments will cause the precise colors to vary ever so slightly. People can't perceive the difference, but code can. And since it uses a triangle, this long-running campaign has been named "Operation Triangulation".

In any event, the last time we talked about Operation Triangulation was in connection with Kaspersky Labs, because someone had used it to attack a number of the iPhones used by Kaspersky's security researchers. In hindsight, that was probably a dumb thing to do since nothing is going to motivate security researchers more than for them to find something unknown crawling around on their own devices. Recall that it was when I found that my own PC was phoning home to some unknown server that I dug down, discovered the Aureate Spyware, as far as I know, coining that term in the process since these were the very early days, and then I wrote "OptOpt" – the world's first spyware removal tool. My point is, if you want your malware to remain secret and thus useful in the long-term, you need to be careful about whose devices you infect.

Though we haven't talked about Kaspersky and the infection of their devices for some time, it turns out that they never stopped working to get to the bottom of what was going on... and they finally have. What they found is somewhat astonishing. And even though it leaves us with some very troubling and annoying unanswered questions, which conspiracy theorists are already having a field day with, what has been learned needs to be shared and understood because thanks to Kaspersky's dogged research we now know everything about the "What" – even if the "How" and the "Why" will probably forever remain unknown. And there's even the chance that parts of this will forever remain unknown to Apple themselves.

Kaspersky's researchers affirmatively and without question found a deliberately concealed, never documented, deliberately locked but unlockable with a secret hash, hardware backdoor which was designed into all Apple devices starting with the A12, A13, A14, A15 and A16.

This now publicly known backdoor has been given the CVE which is today's podcast title, thus CVE-2023-38606. Though it's really not clear to me that it should be a CVE since it's not a bug, it's a deliberately designed-in and protected feature. Regardless, if we call **it** a 0-day, then it's one of four 0-days which, when used in a sophisticated attack chain, along with three other 0-days, is being described as the most sophisticated attack ever discovered against Apple devices, and that's a characterization I would concur with.

Okay... so let's back up a bit and look at what we know thanks to Kaspersky's work. I can't wait to tell everyone about 38606, but to understand its place in the overall attack we need to put it into context.

The world at large learned of all this just last Wednesday, on December 27th, when a team from Kaspersky presented and detailed their findings during the 37th, 4-day, Chaos Communication Congress held at the Congress Center in Hamburg, Germany. The title they gave their presentation was "Operation Triangulation: What You Get When Attack iPhones of Researchers", which, I think, is a perfect title. On the same day last Wednesday they also posted a non-presentation description of their research on their own blog, titled: "Operation Triangulation: The last (hardware) mystery".

https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/

I've edited their lengthy posting for the podcast but I wanted to retain the spirit of their disclosure since I think they got it all just right and they did not venture into conspiracies. So, with some editing for clarity and length, here's what they explained:

Today, on December 27, 2023, we delivered a presentation, titled, "Operation Triangulation: What You Get When Attack iPhones of Researchers", at the 37th Chaos Communication Congress (37C3), held at Congress Center Hamburg. The presentation summarized the results of our long-term research into Operation Triangulation, conducted with our colleagues.

This presentation was also the first time we had publicly disclosed the details of all exploits and vulnerabilities that were used in the attack. We discover and analyze new exploits and attacks such as these on a daily basis. We have discovered and reported more than 30 in-the-wild zero-days in Adobe, Apple, Google, and Microsoft products ... but this is definitely the most sophisticated attack chain we have ever seen.

Here is a quick rundown of this 0-click iMessage attack, which used four zero-days and was designed to work on iOS versions up to iOS 16.2.

- *Attackers send a malicious iMessage attachment, which the application processes without showing any signs to the user.*
- *This attachment exploits the remote code execution vulnerability CVE-2023-41990 in the undocumented, Apple-only ADJUST TrueType font instruction. This instruction had existed since the early 90's until a patch removed it.*
- *It uses return/jump oriented programming and multiple stages written in the NSExpression/NSPredicate query language, patching the JavaScriptCore library environment to execute a privilege escalation exploit written in JavaScript.*
- *This JavaScript exploit is obfuscated to make it completely unreadable and to minimize its size. Still, it has around 11,000 lines of code, which are mainly dedicated to JavaScriptCore and kernel memory parsing and manipulation.*
- *It exploits the JavaScriptCore debugging feature DollarVM ($vm) to gain the ability to manipulate JavaScriptCore's memory from the script and execute native API functions.*
- *It was designed to support both old and new iPhones and included a Pointer Authentication Code (PAC) bypass for exploitation of recent models.*
- *It uses the integer overflow vulnerability CVE-2023-32434 in XNU's memory mapping syscalls to obtain read/write access to the entire physical memory of the device.*

- *It uses hardware memory-mapped I/O (MMIO) registers to bypass the Page Protection Layer (PPL). This was mitigated as CVE-2023-38606.*

- *After exploiting all the vulnerabilities, the JavaScript exploit can do whatever it wants to the device including running spyware. But the attackers chose to: (a) launch the IMAgent process and inject a payload that clears the exploitation artifacts from the device; (b) run a Safari process in invisible mode and forward it to a web page with the next stage.*

- *The web page has a script that verifies the victim and, if the checks pass, receives the next stage: the Safari exploit.*

- *The Safari exploit uses CVE-2023-32435 to execute a shellcode.*

- *The shellcode executes another kernel exploit in the form of a Mach object file. The shellcode reuses the previously used vulnerabilities: CVE-2023-32434, CVE-2023-38606. It is also massive in terms of size and functionality, but completely different from the kernel exploit written in JavaScript. Certain parts related to exploitation of the above- mentioned vulnerabilities are all that the two share. Still, most of its code is also dedicated to parsing and manipulation of the kernel memory. It contains various post-exploitation utilities, which are mostly unused.*

- *The exploit obtains root privileges and proceeds to execute other stages, which load spyware.*

So the view we have from 10,000 feet is of an extremely potent and powerful attack chain which, unbeknownst to any targeted iPhone user, arranges to load, in sequence, a pair of extremely powerful and flexible attack kits. The first of the kits works to immediately remove all artifacts of its presence to erase any trace of what it is and how it got there. It also triggers the execution of the second extensive attack kit which obtains root privileges on the device and then loads whatever subsequent spyware the attackers have selected.

So it uses CVE 41990, a remote code execution vulnerability in the undocumented, Apple-only ADJUST TrueType font instruction. With that still somewhat limited but useful capability, it uses "living off the land" return/jump oriented programming, meaning that since it cannot really bring much code along with it, it jumps to the ends of existing Apple-supplied code subroutines, threading that together to patch the JavaScriptCore library just enough to obtain obtain privilege escalation. By using the library's debugging features, it's able to execute native Apple API calls with privilege. It then uses another vulnerability (CVE 32434), an integer overflow vulnerability in the operating system memory mapping system API, to obtain read/write access to the entire physical memory of the device. And read/write access to physical memory is required for the exploitation of CVE 38606, which is used to disable the entire system's write protection.

So far we've used three vulnerabilities: 41990, 32434 and 38606. The big deal is that by arranging to get to 38606 to work, which requires read/write access to physical memory, because that's how it's controlled, this exploit arranges to completely disable Apple's Page Protection Layer (PPL) which is what provides a great deal of the modern lock-down of iOS devices. Here's how Apple describes their PPL:

Page Protection Layer (PPL) *in iOS, iPadOS, and watchOS is designed to prevent user space code from being modified after code signature verification is complete. Building on Kernel Integrity Protection and Fast Permission Restrictions, PPL manages the page table permission overrides to make sure only the PPL can alter protected pages containing user code and page tables. The system provides a massive reduction in attack surface by supporting systemwide code integrity enforcement, even in the face of a compromised kernel. This protection isn't offered in macOS because PPL is only applicable on systems where all executed code must be signed.*

The key sentence there was: "The system provides **a massive reduction in attack surface** by supporting systemwide code integrity enforcement, even in the face of a compromised kernel." So that means that defeating the PPL protections results in a massive increase in attack surface.

Once those first three chained, and cleverly deployed vulnerabilities have been leveraged, as Kaspersky puts it: *"the JavaScript exploit can do whatever it wants to the device including running spyware."* In other words, the targeted iDevice has been torn wide open.

The next thing Kaspersky does is to take a closer look at the subject of their posting and of this podcast. But before we look at that I need to explain a bit about the concept of memory mapped IO. From the beginning, computers used separate input and output instructions to read and write from their peripheral devices, and read and write instructions to read from and write to their main memory. This idea of having separate instructions for communicating with devices versus loading and storing to and from memory seemed so intuitive that it probably outlasted its usefulness. What this really meant was that IO and memory occupied separate address spaces. Processors would place the address their code was interested in on the system's address bus. Then, if the IO_READ signal was pulsed that address would be taken as the address of some peripheral device's register. But if the MEMORY_READ signal was pulsed, the same bus would be used to address and read from main memory. To this day, largely for the sake of backward compatibility, Intel processors have separate input and output instructions that operate like this, but they are rarely used any longer.

What happened was that someone realized that if a peripheral device's hardware registers were actually addressable just like regular memory, meaning **in** the system's main memory address space right alongside the rest of actual memory, then the processor's design could be simplified by eliminating separate input and output instructions and all of the already existing instructions for reading and writing to and from memory could perform double duty as IO instructions.

To be practical, one of the requirements is that the system needs to have plenty of spare memory address bits, but even a 32-bit system, which can uniquely address 4.3 billion bytes of RAM can spare some space for its IO. And that's exactly what has transpired. When today's SpinRite wishes to check on the status of one of the system's hard drives, it reads that drive's status from a memory address up near the top of the system's 32-bit address space. Even though it's reading a memory address, the data that's read is actually coming from the drive's hardware. This is known as memory mapped IO because the system's IO is mapped into the processor's memory address space and discrete input and output instructions are no longer used.

Today's massive systems — and by "massive systems" I mean an iPhone because it has grown into a massive and complex system — use 64-bit chips with ridiculously large memory spaces. So all of the great many various functions of the system are reflected in a huge and sparsely populated memory map.

In the past, we've talked about the limited size of the 32-bit IPv4 address space and how it's no longer possible to hide in IPv4 because it's just not big enough. The entire IPv4 address space is routinely being scanned. This is not true for IPv6 with its massive 128-bit addressing space. Unlike IPv4 it's not possible to exhaustively scan IPv6 space – there's just too much of it. So here's my point: When there is a ton of unused space it's possible to leave undocumented some of the functions of memory mapped peripherals and there's really no way for code to know whether anything might be listening at any given address or not. And not only that, there may be too many addresses to reasonably check.

What Kaspersky discovered was that Apple's hardware chips, from the A12 through the A16 all incorporated exactly this sort of hardware backdoor. And get this: This deliberately designed-in backdoor even incorporates a secret hash. In order to use it, the software must run a custom (not very secure but still secret) hash function to essentially sign the request that it's submitting to this backdoor.

Here's some of what the Kaspersky guys wrote about their discovery. And note that even they did not discover it – because it's explicitly not discoverable. They discovered its use by malware which they were reverse-engineering. The question that has the conspiracy folks all wound up is "how did non-Apple bad guys discover it?" In the section of their paper titled "The mystery of the CVE-2023-38606 vulnerability" Kaspersky wrote:

> *What we want to discuss is related to the vulnerability that has been mitigated as CVE-2023-38606. Recent iPhone models have additional hardware-based security protection for sensitive regions of the kernel memory. This protection prevents attackers from obtaining full control over the device if they can read and write kernel memory, as achieved in this attack by exploiting CVE-2023-32434. We discovered that to bypass this hardware-based security protection, the attackers used another hardware feature of Apple-designed SoCs (systems on chip).*
>
> *If we try to describe this feature and how the attackers took advantage of it, it all comes down to this: they are able to write data to a certain physical address while bypassing the hardware-based memory protection by writing the data, destination address, and data hash to unknown hardware registers of the chip unused by the firmware.*
>
> *Our guess is that this unknown hardware feature was most likely intended to be used for debugging or testing purposes by Apple engineers or the factory, or that it was included by mistake. Because this feature is not used by the firmware, we have no idea how attackers would know how to use it.*
>
> *We are publishing the technical details, so that other iOS security researchers can confirm our findings and come up with possible explanations of how the attackers learned about this hardware feature.*

Various peripheral devices available in the System on Chip may provide special hardware registers that can be used by the CPU to operate these devices. For this to work, these hardware registers are mapped to the memory accessible by the CPU and are known as "memory-mapped I/O (MMIO)".

Address ranges for MMIOs of peripheral devices in Apple products (iPhones, Macs, and others) are stored in a special file format: DeviceTree. Device tree files can be extracted from the firmware, and their contents can be viewed with the help of the dt utility.

While analyzing the exploit used in the Operation Triangulation attack, I discovered that most of the MMIOs used by the attackers to bypass the hardware-based kernel memory protection do not belong to any MMIO ranges defined in the device tree. The exploit targets Apple A12–A16 Bionic SoCs, targeting unknown MMIO blocks of registers which we have documented.

This prompted me to try something. I checked different device tree files for different devices and different firmware files: no luck. I checked publicly available source code: no luck. I checked the kernel images, kernel extensions, iboot, and coprocessor firmware in search of a direct reference to these addresses: nothing.

How could it be that the exploit used MMIOs that were not used by the firmware? How did the attackers find out about them? What peripheral device(s) do these MMIO addresses belong to? It occurred to me that I should check what other known MMIOs were located in the area close to these unknown MMIO blocks. That approach was successful and I discovered that the memory ranges used by the exploit surrounded the system's GPU coprocessor. This suggested that all these MMIO registers most likely belonged to the GPU coprocessor!

After that, I took a closer look at the exploit and found one more thing that confirmed my theory. The first thing the exploit does during initialization is write to some other MMIO register, which is located at a different address for each version of Apple's System on Chip.

```
if (cpuid == 0x8765EDEA):   # CPUFAMILY_ARM_EVEREST_SAWTOOTH (A16)
    base = 0x23B700408
    command = 0x1F0023FF

elif (cpuid == 0xDA33D83D): # CPUFAMILY_ARM_AVALANCHE_BLIZZARD (A15)
    base = 0x23B7003C8
    command = 0x1F0023FF

elif (cpuid == 0x1B588BB3): # CPUFAMILY_ARM_FIRESTORM_ICESTORM (A14)
    base = 0x23B7003D0
    command = 0x1F0023FF

elif (cpuid == 0x462504D2): # CPUFAMILY_ARM_LIGHTNING_THUNDER (A13)
    base = 0x23B080390
    command = 0x1F0003FF

elif (cpuid == 0x07D34B9F): # CPUFAMILY_ARM_VORTEX_TEMPEST (A12)
    base = 0x23B080388
    command = 0x1F0003FF

if ((~read_dword(base) & 0xF) != 0):
    write_dword(base, command)
    while(True):
        if ((~read_dword(base) & 0xF) == 0):
            break
```

Kaspersky's posting shows some pseudocode for what they found the doing. Each of the five different Apple Bionic chips A12 through A16 is identified by a unique CPU ID which is readily available to code. So the code looks up the CPU ID, then chooses a custom per-processor address and command, which is different for each chip generation, which it then uses to unlock this undocumented feature of Apple's recent chips.

> *With the help of the device tree and Siguza's utility, pmgr, I was able to discover that all these addresses corresponded to the GFX register in the power manager MMIO range.*
>
> *Finally, I obtained a third confirmation when I decided to try and access the registers located in these unknown regions. Almost instantly, the GPU coprocessor panicked with a message of, "GFX ERROR Exception class=0x2f (Error interrupt), IL=1, iss=0 – power(1)".*
>
> *This way, I was able to confirm that all these unknown MMIO registers used for the exploitation belonged to the GPU coprocessor. This motivated me to take a deeper look at its firmware, which is written in ARM code and unencrypted, but I could not find anything related to these registers in there.*
>
> *I decided to take a closer look at how the exploit operated these unknown MMIO registers. One register (located at hex 206040000) stands out from all the others because it is located in a separate MMIO block from the other registers. It is touched only during the initialization and finalization stages of the exploit: it is the first register to be set during initialization and the last one, during finalization. From my experience, it was clear that the register either enabled or disabled the hardware feature used by the exploit or controlled interrupts. I started to follow the interrupt route, and fairly soon, I was able to recognize this unknown register, and also discovered what exactly was mapped to the address range containing it.*

Through additional reverse-engineering of the exploit and watching it function on their devices, the Kaspersky guys were able to work out exactly what was going on. The exploit provides the address for a Direct Memory Access write, the data to be written and a custom hash which signs the block of data to be written. The hash signature serves to prove the whomever is doing this has access to super-secret knowledge that only Apple would possess. This authenticates the validity of the request for the data to be written.

And speaking of the hash signature, this is what Kaspersky wrote:

> *Now that all the work with all the MMIO registers has been covered, let us take a look at one last thing: how hashes are calculated. The algorithm is shown and as you can see, it is a custom algorithm, with the hash is calculated by using a predefined SBox table. I tried to search for it in a large collection of binaries, but found nothing. You may notice that this hash does not look very secure, as it occupies just 20 bits (10+10, as it is calculated twice), but it does its job as long as no one knows how to calculate and use it. It is best summarized with the term "security by obscurity". How could attackers discover and exploit this hardware feature if it is never used — and there are no instructions anywhere — in the firmware on how to use it?*

In crypto parlance, an "SBox" is simply a lookup table, typically an 8-bit lookup of 256 pseudo random but unchanging values. The "S" of SBox stands for substitution. SBoxes are widely used in crypto and hashes because they provide a very fast means for mapping one byte into another

in a completely arbitrary way. The show notes shows the SBox and the very simple lookup and XOR algorithm that's used by the exploit:

```
sbox = [
    0x007, 0x00B, 0x00D, 0x013, 0x00E, 0x015, 0x01F, 0x016,
    0x019, 0x023, 0x02F, 0x037, 0x04F, 0x01A, 0x025, 0x043,
    0x03B, 0x057, 0x08F, 0x01C, 0x026, 0x029, 0x03D, 0x045,
    0x05B, 0x083, 0x097, 0x03E, 0x05D, 0x09B, 0x067, 0x117,
    0x02A, 0x031, 0x046, 0x049, 0x085, 0x103, 0x05E, 0x09D,
    0x06B, 0x0A7, 0x11B, 0x217, 0x09E, 0x06D, 0x0AB, 0x0C7,
    0x127, 0x02C, 0x032, 0x04A, 0x051, 0x086, 0x089, 0x105,
    0x203, 0x06E, 0x0AD, 0x12B, 0x147, 0x227, 0x034, 0x04C,
    0x052, 0x076, 0x08A, 0x091, 0x0AE, 0x106, 0x109, 0x0D3,
    0x12D, 0x205, 0x22B, 0x247, 0x07A, 0x0D5, 0x153, 0x22D,
    0x038, 0x054, 0x08C, 0x092, 0x061, 0x10A, 0x111, 0x206,
    0x209, 0x07C, 0x0BA, 0x0D6, 0x155, 0x193, 0x253, 0x28B,
    0x307, 0x0BC, 0x0DA, 0x156, 0x255, 0x293, 0x30B, 0x058,
    0x094, 0x062, 0x10C, 0x112, 0x0A1, 0x20A, 0x211, 0x0DC,
    0x196, 0x199, 0x256, 0x165, 0x259, 0x263, 0x30D, 0x313,
    0x098, 0x064, 0x114, 0x0A2, 0x15C, 0x0EA, 0x20C, 0x0C1,
    0x121, 0x212, 0x166, 0x19A, 0x299, 0x265, 0x2A3, 0x315,
    0x0EC, 0x1A6, 0x29A, 0x266, 0x1A9, 0x269, 0x319, 0x2C3,
    0x323, 0x068, 0x0A4, 0x118, 0x0C2, 0x122, 0x214, 0x141,
    0x221, 0x0F4, 0x16C, 0x1AA, 0x2A9, 0x325, 0x343, 0x0F8,
    0x174, 0x1AC, 0x2AA, 0x326, 0x329, 0x345, 0x383, 0x070,
    0x0A8, 0x0C4, 0x124, 0x218, 0x142, 0x222, 0x181, 0x241,
    0x178, 0x2AC, 0x32A, 0x2D1, 0x0B0, 0x0C8, 0x128, 0x144,
    0x1B8, 0x224, 0x1D4, 0x182, 0x242, 0x2D2, 0x32C, 0x281,
    0x351, 0x389, 0x1D8, 0x2D4, 0x352, 0x38A, 0x391, 0x0D0,
    0x130, 0x148, 0x228, 0x184, 0x244, 0x282, 0x301, 0x1E4,
    0x2D8, 0x354, 0x38C, 0x392, 0x1E8, 0x2E4, 0x358, 0x394,
    0x362, 0x3A1, 0x150, 0x230, 0x188, 0x248, 0x284, 0x302,
    0x1F0, 0x2E8, 0x364, 0x398, 0x3A2, 0x0E0, 0x190, 0x250,
    0x2F0, 0x288, 0x368, 0x304, 0x3A4, 0x370, 0x3A8, 0x3C4,
    0x160, 0x290, 0x308, 0x3B0, 0x3C8, 0x3D0, 0x1A0, 0x260,
    0x310, 0x1C0, 0x2A0, 0x3E0, 0x2C0, 0x320, 0x340, 0x380
]

def calculate_hash(buffer):

    acc = 0
    for i in range(8):
        pos = i * 4
        value = read_dword(buffer + pos)
        for j in range(32):
            if (((value >> j) & 1) != 0):
                acc ^= sbox[32 * i + j]

    return acc
```

The $64,000 question here is: How could anyone outside of Apple possibly obtain exactly this 256-entry lookup table, which is required to create the hash that signs the request? Some have suggested deep hardware reverse-engineering, popping the lid off of the chip and tracing its circuitry? But because this is a known avenue of general vulnerability, chips are well protected from this form of reverse engineering. And that sure seems like a stretch in any event. Every one of the five processor generations had this same backdoor, differing only in the enabling command and address.

Some have suggested that this was implanted into Apple's devices without their knowledge and that the disclosure of this would have come as a horrible surprise to them. But that seems far-fetched to me as well. Again, largely identical implementations with a single difference across five generations of processor.

And then to have the added **protection** against its use or coincidental discovery of requiring a hash signature for every request. To me, that's the mark of Apple's hand in this. If this was implanted without Apple's knowledge, the implant would have been as small and innocuous as possible, and the implanter would have been less concerned about its misuse, satisfied with the protection that it was unknown. Add to that the fact that the address space this uses is wrapped

into and around the region used by the GPU coprocessor. This suggests that they had to be designed in tandem.

After Apple was informed that knowledge of their little hardware backdoor had somehow escaped into the wild and was being abused as part of the most sophisticated attack on iPhones ever seen, they quickly mitigated this vulnerability with the release of iOS 16.6. During its boot up, the updated OS unmaps access to the required memory ranges through their processor's memory manager. This removes the logical to physical address mapping and access that any code subsequently running inside the chip would need.

The Kaspersky guys conclude by writing:

> *This is no ordinary vulnerability, and we have many unanswered questions. We do not know how the attackers learned to use this unknown hardware feature or what its original purpose was. Neither do we know if it was developed by Apple or it's a third-party component like ARM CoreSight.*
>
> *What we do know—and what this vulnerability demonstrates—is that advanced hardware-based protections are useless in the face of a sophisticated attacker as long as there are hardware features that can bypass those protections.*
>
> *Hardware security very often relies on "security through obscurity", and it is much more difficult to reverse-engineer than software, but this is a flawed approach, because sooner or later, all secrets are revealed. Systems that rely on "security through obscurity" can never be truly secure.*

Kaspersky's conclusions here … are wrong. The Kaspersky guys are clearly reverse engineering geniuses; and Apple, who certainly never wanted knowledge of this backdoor to fall into the hands of the underworld for use by hostile governments and nationstates, owes them big time for figuring out that this closely held secret had somehow escaped. But I take issue with Kaspersky's use of the overused phrase **"security through obscurity."** That doesn't apply here. The term "obscure" suggests discoverability. That's why something that's obscure and thus discoverable is not truly secure... **because it could be discovered.** But what Kaspersky found could **not** be discovered – by anyone – not conceivably. Because it was protected by a complex **secret** hash. Kaspersky themselves did not discover it. They watched the malware, which had this knowledge, use this secret feature and only then figured out what must be going on. And that 256-entry hash table which Kaspersky documented came from inside the malware which had obtained it from somewhere else — from someone who knew the secret.

So let's get this very clear because it's an important point: There is nothing whatsoever **obscure** about this. The use of this backdoor required a priori knowledge — explicit knowledge in advance of its use. And that knowledge had to come from whatever entity implemented this. Period.

We also need to note that as powerful as this backdoor was, access to it first required access to its physical memory-mapped IO range which is, and was, explicitly unavailable. Software running on these iDevices had no access to this protected hardware address space. So it wasn't as if anyone who knew this secret could just waltz in and have their way with any Apple device. There were still several layers of preceding protection which needed to be bypassed.

And that was the job of the two 0-day exploits which preceded the use of 38606.

So we're left with the question, why is this very powerful and clearly meant-to-be-kept-secret facility present in the past five generations of Apple's processors?

I don't buy the argument that this is some sort of debugging facility that was left in by mistake. For one thing, it was there with subtle changes through five generations of chips. That doesn't feel like a mistake; and there are far more straightforward means for debugging. No one hash-signs a packet to be written by DMA into memory while debugging. That makes no sense. Also, the fact that its use is guarded by a secret hash reveals and proves that **it was intended to be enabled and present in the wild.** The hash forms a secret key that explicitly allows this to exist safely and without fear of malicious exploitation without the explicit knowledge of the hash function. This is a crucial point that I want to be certain everyone appreciates. This was clearly meant to be active in the wild with its abuse being quite well protected by a well-kept secret.

So again, why was it there? Unless Apple fesses up, we'll likely never know.

Could this backdoor system have always been part of some large ARM silicon library that Apple licensed without ever taking a close look at it? That doesn't really sound like the hands-on Apple we all know. But it's been suggested that this deliberately engineered backdoor technology might have somehow been present through five generations of Apple's custom silicon without Apple ever being aware of it. One reason that idea falls flat is that Apple's own firmware Device Tree, which provides the mapping of all peripheral hardware into memory, accommodates these undocumented memory ranges without any description. Unlike any other known component, nowhere are these memory ranges described. No purpose is given. And Apple's patch for this, changes the descriptors for those memory ranges to "DENY" to prevent their future access.

The other question, since the hash function is not **obscure**, it is truly **secret**, is how did this **secret** come to be in the hands of those who could exploit it for their own ends? No matter why Apple had this present in their systems, that could never have been their intent. And here I agree with the Kaspersky guys when they say: *"... this is a flawed approach, because sooner or later, all secrets are revealed."* Somewhere, people within Apple knew of this backdoor. They knew that this backdoor was present and they knew how to access it. And somehow that secret escaped from Apple's control.

What we do know is that for the time being at least, iDevices are **more** secure than they have been, because backdoor access through **this** means has been removed, after it became public.

But because we don't know why this was present in the first place, we should have no confidence that next-generation Apple silicon won't have another similar feature present. Its memory-mapped IO hardware location would be changed, its hash function would be different, and Apple will likely have learned something from this debacle about keeping this closely held secret even more closely held.

Do they have a secret deal with the NSA to allow their devices to be backdoored? Those prone to secret conspiracies can only speculate. And even so, it does take access to additional secrets or vulnerabilities to access that locked backdoor. So using it is not easy.

I would prefer to believe that Apple was an unwitting accomplice in this. That this was an ultra sophisticated supply chain attack. That some agency influenced the upstream design of modules that Apple acquired and simply dropped into place without examination. And that it was this agency who therefore had and held the secret hash function and knew how to use it. As they say, anything's possible. But that stretches credulity farther than is probably warranted. Occam's Razor suggests that the simplest explanation is the most likely to be correct. And the simplest explanation here is that Apple deliberately built a secret backdoor into the past five most recent generations of their silicon.

One thing that should be certain, is that Apple has lost a bit of its security shine with this one. A bit of erosion of trust is clearly in order, because five generations of Apple silicon contained a backdoor that was protected by a secret hash whose only reasonable purpose could have been to allow its use to be securely made available, after sale ... in the wild. And that that leaves us with one question:

Why?