God Mode On

Researchers run Doom on a vehicle's head unit after remotely attacking its modem

Alexander Kozlov
Sergey Anufrienko

20.11.2025 Version 1.0

Exploiting a vulnerability identified in a modem installed in the head units of some vehicles enabled Kaspersky ICS CERT experts to gain complete control of the system.

Introduction

Imagine you are a driver speeding down the highway in your brand-new electric car. All of a sudden, the entire massive multimedia display is filled with Doom, the iconic 3D shooter game, replacing the navigation map or the controls menu, and you realize someone is playing it right now by remotely controlling the character. This is not a dream or an overactive imagination, but a realistic scenario in today's world, as vividly demonstrated by Kaspersky ICS CERT experts.

We know that the Internet of Things plays a significant role in the modern world, where not only smartphones and laptops, but also factories, cars, trains, and even airplanes are connected to the network. Most of the time, connectivity is provided via 3G/4G/5G mobile data networks using modems installed in these vehicles and devices. Increasingly, these modems are integrated into a System-on-Chip (SoC), which can simultaneously perform multiple functions using a Communication Processor (CP) and an Application Processor (AP). A general-purpose operating system such as Android can run on the AP, while the CP, which handles communication with the mobile network, typically runs on a dedicated OS. The interaction between the AP, CP, and RAM within the SoC at the microarchitecture level is a "black box" known only to the manufacturer – even though the security of the entire SoC depends on it.

It is generally believed that bypassing 3G/LTE security mechanisms is purely an academic challenge, because a secure communication channel is established when a user device (User Equipment, UE) connects to a cellular base station (Evolved Node B, eNB). Even if someone can bypass these mechanisms, discover a vulnerability in the modem, and execute their own code on it, this is unlikely to compromise the device's business logic. This logic (for example, user applications, browser history, calls, and SMS on a smartphone) resides on the AP and is presumably not accessible from the modem. Or is it?

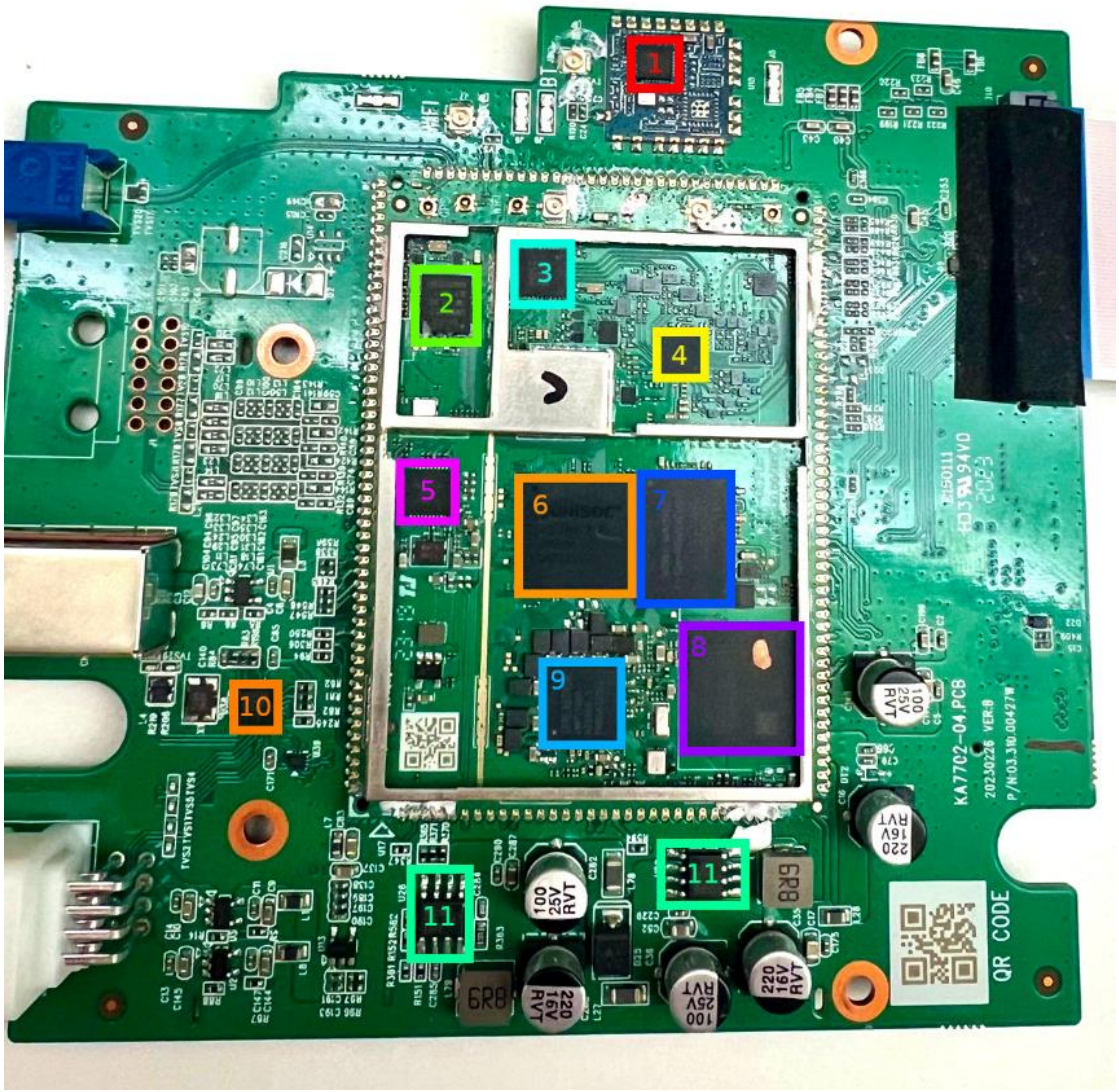
To find out, we conducted a security assessment of a modern SoC, Unisoc UIS7862A, which features an integrated 2G/3G/4G modem. This SoC can be found in various mobile devices by multiple vendors or, more interestingly, in the head units of modern Chinese vehicles, which are becoming increasingly common on the roads. The head unit is one of a car's key components, and a breach of its information security poses a threat to road safety, as well as the confidentiality of user data.

During our research, we identified several critical vulnerabilities at various levels of the Unisoc UIS7862A modem's cellular protocol stack. This article discusses a stack-based buffer overflow vulnerability in the 3G RLC protocol implementation (CVE-2024-39432). The vulnerability can be exploited to achieve remote code execution at the early stages of connection, before any protection mechanisms are activated.

Importantly, gaining the ability to execute code on the modem is only the entry point for a complete remote compromise of the entire SoC. Our subsequent efforts were focused on gaining access to the AP. We discovered several ways to do so, including leveraging a hardware vulnerability in the form of a hidden peripheral Direct Memory Access (DMA) device to perform lateral movement within the SoC. This enabled us to install our own patch into the running Android kernel and execute arbitrary code on the AP with the highest privileges. Details are provided in the relevant sections.

Acquiring the modem firmware

The modem at the center of our research was found on the circuit board of the head unit in a Chinese car.



The circuit board of the head unit

Table. Description of the circuit board components

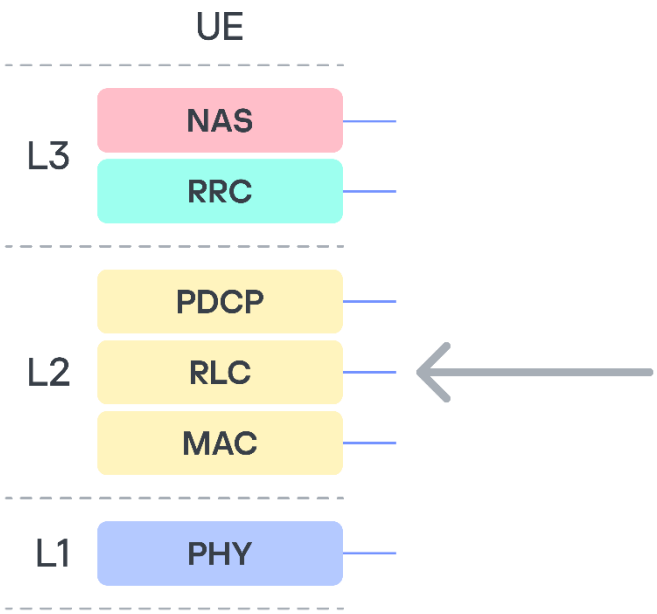
Number in the board photo	Component
1	Realtek RTL8761ATV 802.11b/g/n 2.4G controller with wireless LAN (WLAN) and USB interfaces (USB 1.0/1.1/2.0 standards)
2	SPRD UMW2652 BGA WiFi chip
3	55966 TYADZ 21086 chip
4	SPRD SR3595D (Unisoc) radio frequency transceiver
5	Techpoint TP9950 video decoder
6	UNISOC UIS7862A
7	BIWIN BWSRGX32H2A-48G-X internal storage, Package200-FBGA, ROM Type – Discrete, ROM Size – LPDDR4X, 48G
8	SCY E128CYNT2ABE00 EMMC 128G/JEDEC memory card
9	SPREADTRUM UMP510G5 power controller
10	FEI.1s LE330315 USB2.0 shunt chip
11	SCT2432STER synchronous step-down DC-DC converter with internal compensation

Using information about the modem's hardware, we desoldered and read the embedded multimedia memory card, which contained a complete image of its operating system. We then analyzed the image obtained.

Remote access to the modem (CVE-2024-39431)

The modem under investigation, like any modern modem, implements several protocol stacks: 2G, 3G, and LTE. Clearly, the more protocols a device supports, the more potential entry points (attack vectors) it has. Moreover, the lower in the OSI network model stack a vulnerability sits, the more severe the consequences of its exploitation can be. Therefore, we decided to analyze the data packet fragmentation mechanisms at the data link layer (RLC protocol).

We focused on this protocol because it is used to establish a secure encrypted data transmission channel between the base station and the modem, and, in particular, it is used to transmit NAS (Non-Access Stratum) protocol data. This means that a remote code execution (RCE) vulnerability would allow an attacker to execute their own code on the modem, bypassing all existing 3G communication protection mechanisms.



3G protocol stack

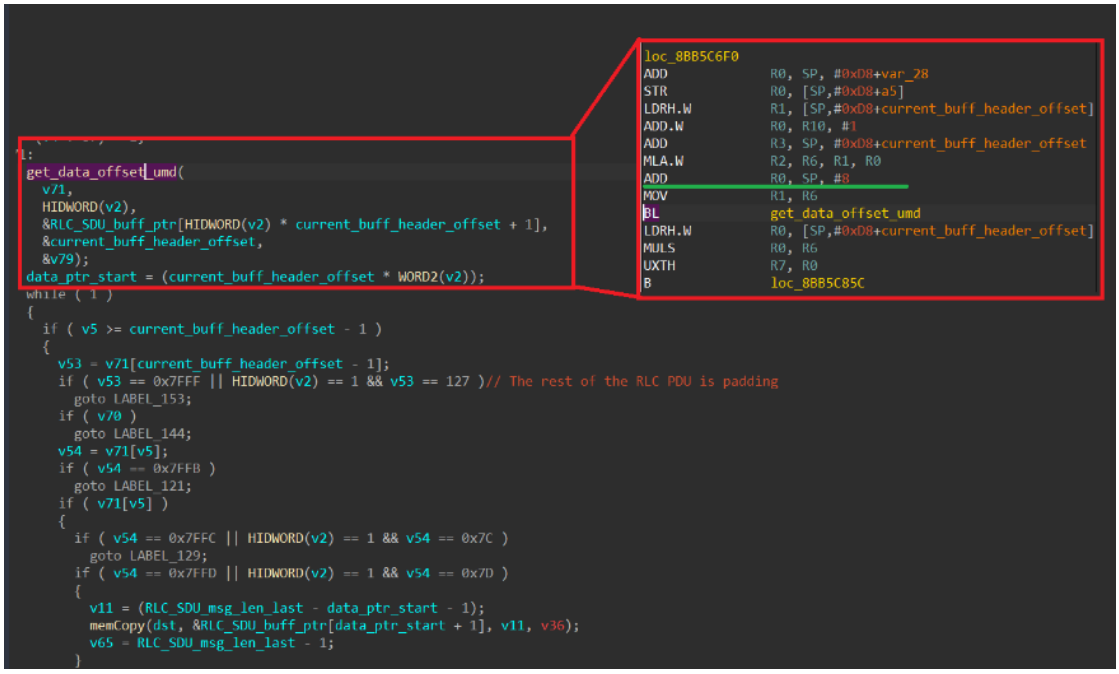
The RLC protocol uses three different transmission modes: TM, UM, and AM. We are only interested in UM (Unacknowledged Mode), because in this mode the 3G standard allows both the segmentation of data and the concatenation of several small higher-layer data fragments (Protocol Data Units, PDU) into a single data link layer frame. This is done to maximize channel utilization. At the RLC level, packets are referred to as Service Data Units (SDU).[1]

To locate SDU handling functions in the firmware among the vast number of different functions (about 75,000), it is sufficient to search for constants used in the process (0x7FFF, 0x7FFC, 0x7FFB). Among all occurrences of these constants, we identify those where they are meaningfully applied in the context of code execution. This yields a limited set of functions, including the function for handling an incoming SDU packet.

When handling an SDU packet received, its header fields are parsed. The packet itself consists of a mandatory header, optional headers, and data. Processing an optional header involves sequentially traversing each field in the header. This is done using the get_data_offset function. It parses the optional headers, collects the sizes of data fragments inside the data section, and returns a pointer to the data within the SDU packet. Important: according to the modem

[1] See the ETSI TS 136 322 standard.

OS code analysis, information about the size of all concatenated PDUs inside an SDU is stored on the stack!



The algorithm processes each header field sequentially. The end of the optional headers is indicated by the least significant bit (E bit) being equal to 0. If it equals 1, processing continues. During processing, data is written to a variable located on the stack of the calling function. The number of optional headers is not limited. The stack depth is 0xB4 bytes. The size of the packet that can be parsed (i.e., the number of headers, each header being a 2-byte entry on the stack) is limited by the SDU packet size of 0x5F0 bytes.

As a result, exploitation can be achieved using just one packet in which the number of headers exceeds the stack depth (90 headers). It is important to note

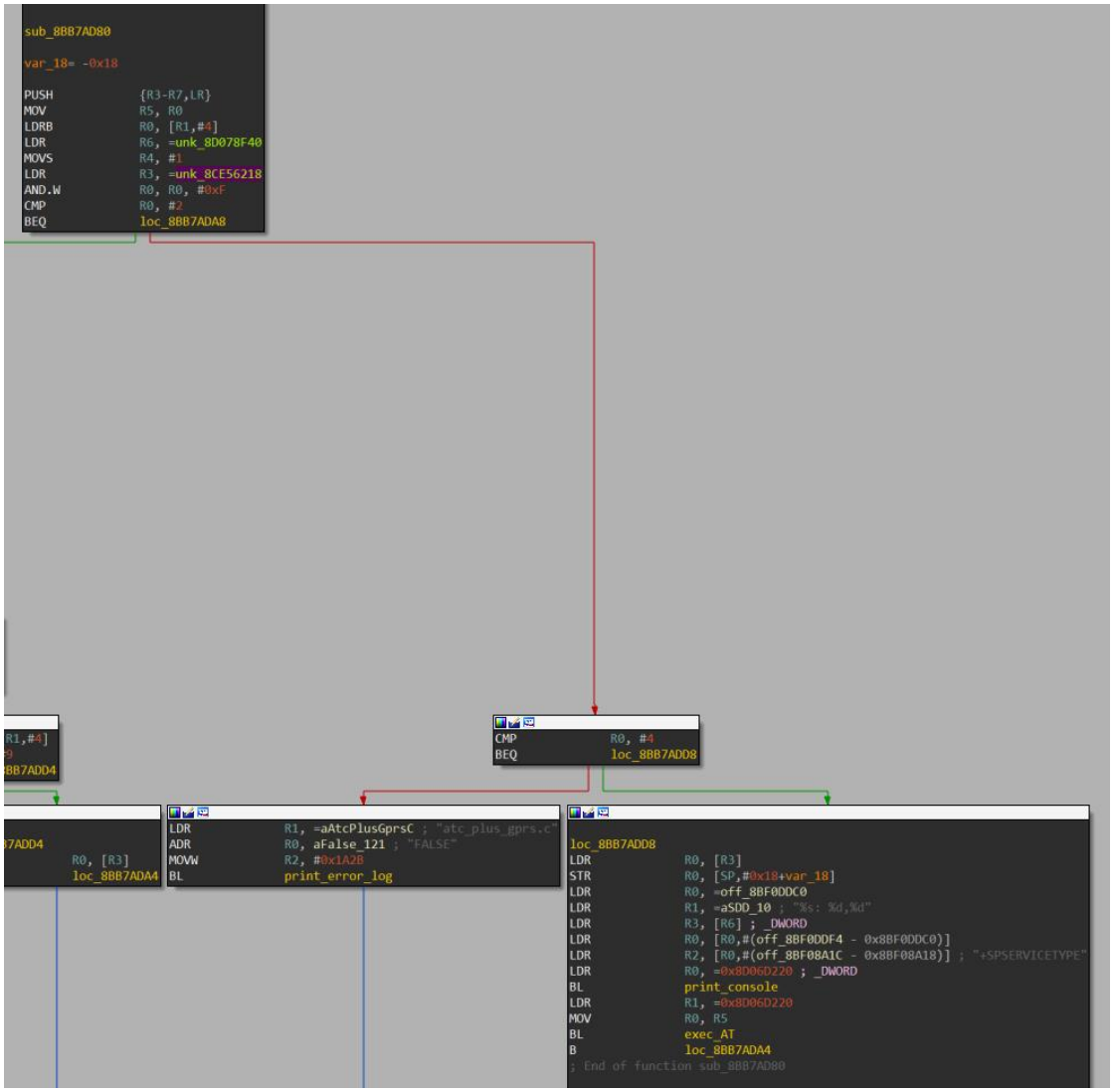
that this particular function lacks a stack canary, and when the stack overflows, it is possible to overwrite the return address and some non-volatile register values in this function. However, overwriting is only possible with a value ending in one in binary (i.e., a value in which the least significant bit equals 1). Notably, execution takes place on ARM in Thumb mode, so all return addresses must have the least significant bit equal to 1. Coincidence? Perhaps.

In any case, sending the very first dummy SDU packet with the appropriate number of "correct" headers caused the device to reboot. However, at that moment, we had no way to obtain information on where and why the crash occurred (although we suspect the cause was an attempt to transfer control to the address 0xAABBCCDD, taken from our packet). Let's see what can be done about this.

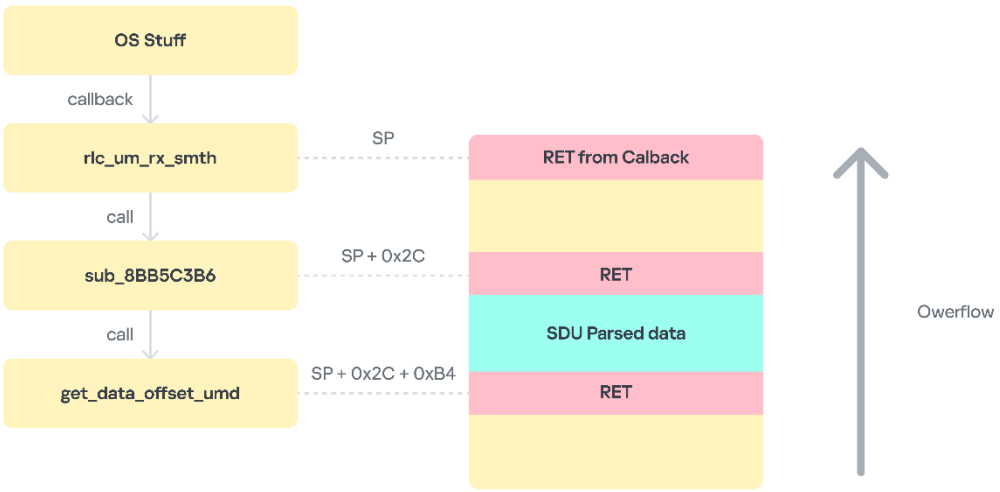
Gaining persistence in the system

The first and most important observation is that we know the pointer to the newly received SDU packet is stored in register R2. Return Oriented Programming (ROP) techniques can be used to execute our own code. However, the question remains: can we actually execute our code?

We will utilize the available AT command handler to ensure that we really can execute our code on the modem side. Since we do not know the current address of the stack frame where our data is located, nor whether the stack is executable at all, and we are unsure if code sections are writable (although we suspect they are not), the most reliable way is to move the data to RAM areas. To do this, we look for a suitable function among the available AT commands. The first appropriate command is SPSERVICETYPE.



Next, ROP gadgets need to be used to overwrite the address 0x8CE56218 without disrupting the subsequent operation of the incoming SDU packet handling algorithm. To achieve this, it is sufficient to return to the function from which the SDU packet handler was called, because it is invoked as a callback, meaning there is no data linkage on the stack. Given that this function only added 0x2C bytes to the stack, we need to fit within this size.



Stack overflow in the context of the operating system

Having found a suitable ROP chain, we launch an SDU packet that contains it as a payload. As a result, we see the output 0xAABBCCDD in the AT command console for SPSERVICETYPE. Our code works!

```
at /dev/sttu_lte5 & sleep 0.1; echo -e "AT+SPSERVICETYPE?\r" > /dev/stty_lte5 *


+SPSERVICETYPE: 0,2864434397


OK
console:/ #
```

Next, by analogy, we input the address of the stack frame where our data is located. Unfortunately, it turns out that the stack is not executable. Now, we face the task of figuring out the MPU settings on the modem. Once again, using the ROP chain method, we generate code that reads the MPU table, one DWORD at a time. After many iterations, we obtain the following table.

```
Base Address: 00 00 00 00   Size and Regs: 00 00 00 3F   ACR: 00 00 10 00 (0x00000000) 4GB
Base Address: 00 00 00 00   Size and Regs: 00 00 00 3D   ACR: 00 00 11 10 (0x00000000) 2GB

Base Address: 00 00 00 00   Size and Regs: 00 00 00 17   ACR: 00 00 00 10 (0x00000000) 4Kb
Base Address: 00 00 00 00   Size and Regs: 00 00 00 13   ACR: 00 00 00 50 (0x00000000) 1Kb
Base Address: 00 00 00 C0   Size and Regs: 00 00 00 13   ACR: 00 00 00 10 (0x000000C0) 1Kb

Base Address: 87 80 00 00   Size and Regs: 00 00 00 2D   ACR: 00 00 00 10 (0x87800000) 8Mb
Base Address: 88 00 00 00   Size and Regs: 00 00 00 33   ACR: 00 00 00 10 (0x88000000) 64Mb
Base Address: 88 00 00 00   Size and Regs: 00 00 E0 31   ACR: 00 00 10 00 (0x88000000) 32Mb (sub-region)
Base Address: 89 40 00 00   Size and Regs: 00 00 00 29   ACR: 00 00 10 00 (0x89400000) 2Mb
Base Address: 8B 00 00 00   Size and Regs: 00 00 00 2F   ACR: 00 00 00 50 (0x8b000000) 16Mb (CODE SECTION)
Base Address: 8C 00 00 00   Size and Regs: 00 00 80 31   ACR: 00 00 00 10 (0x8C000000) 32Mb (sub-region)
Base Address: 8D 80 00 00   Size and Regs: 00 00 F1 2D   ACR: 00 00 00 10 (0x8D800000) 8Mb (sub-region)

Base Address: 00 00 00 00   Size and Regs: 00 00 00 00   ACR: 00 00 00 00 (unused)
Base Address: 00 00 00 00   Size and Regs: 00 00 00 00   ACR: 00 00 00 00 (unused)
Base Address: 00 00 00 00   Size and Regs: 00 00 00 00   ACR: 00 00 00 00 (unused)
Base Address: 00 00 00 00   Size and Regs: 00 00 00 00   ACR: 00 00 00 00 (unused)
```

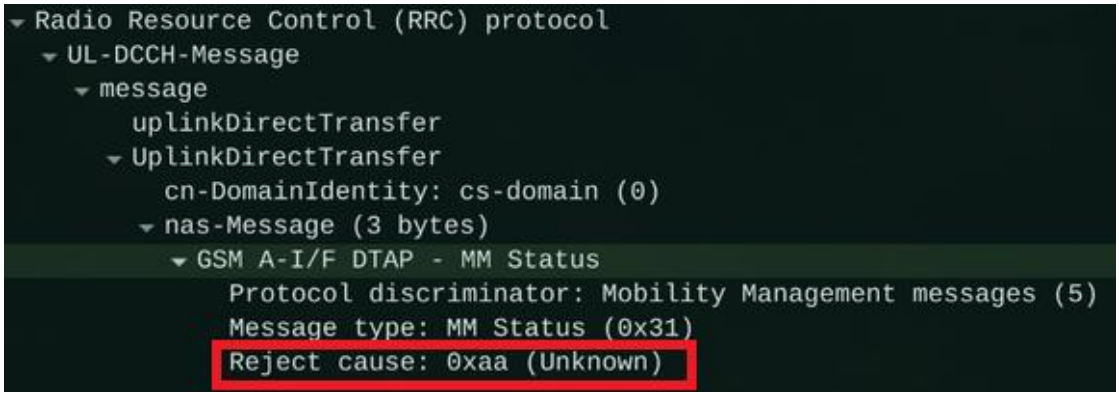
The table shows that, as we suspected, the code section is mapped only for execution. An attempt to change the configuration results in another ROP chain, but this same section is now mapped with write permissions in an unused slot in the table. This is possible due to features of MPU programming, specifically the presence of the overlap mechanism and the fact that a region with a higher ID has higher priority.

```
Base Address: 8B 00 00 00   Size and Regs: 00 00 00 2F   ACR: 00 00 03 08 (0x8B000000) 16Mb
```

All that remains is to use the pointer to our data (which is still stored in R2) and patch the code section that has just been unlocked for writing. The question is what exactly to patch. The simplest way is to patch the NAS protocol handler by adding our code to it. To do this, we use one of the NAS protocol commands – MM information[2]. We can use it to send a large amount of data at once and, in response, receive a single byte of data using the MM status command.

As a result, we not only successfully executed our own code on the modem side but also established full two-way communication with the modem, using the high-level NAS protocol as a means of message delivery. In this case, it is an MM Status packet with the cause field equaling 0xAA.

[2] See the ETSI TS 124 008 standard, p. 393.



However, being able to execute our own code on the modem does not enable us to gain access to user data. Or does it?

Moving laterally within the SoC

An analysis of the internal architecture of the modem processor in the context of the SoC microarchitecture reveals numerous potential attack vectors against the AP. While analyzing the modem's internals, we immediately noticed that it uses physical, not virtual, RAM addresses. Additionally, we noted the addresses at which the AP's operating system kernel resides.

```
ums512_1h10:/data/local/tmp # cat /proc/iomem
.....................................................
70c00000-70c00fff : /soc/ap-apb/i2s@70c00000
70d00000-70d00fff : /soc/ap-apb/i2s@70d00000
70e00000-70e00fff : /soc/ap-apb/i2s@70e00000
71100000-71100fff : /soc/ap-apb/sdio@71100000
71200000-71200fff : /soc/ap-apb/sdio@71200000
71400000-71400fff : /soc/ap-apb/sdio@71400000
71700000-7170ffff : /soc/ap-ahb/vdsp
80000000-1ffffefff : System RAM
  80080000-80d8ffff : Kernel code
  81010000-819cefff : Kernel data
```

This raises a hypothesis: what if the CP and AP share the same address space, since RAM in the SoC is likely implemented as a single hardware component? An analysis of the Device Tree on the AP side further strengthened the assumption that the CP and AP most likely share the same physical address space (see the MPU table provided above).

```
cp-mem at 0x89600000 size 0x4600000
iq-mem at 0x90000000 size 0x4000000
sml-mem at 0x94000000 size 0x20000
debug-mem at 0x100000000 size 0x1000
sipc-mem at 0x87800000 size 0x800000
```

All that remains is to verify this hypothesis. To do so, we patched the MPU table again, this time adding an entry that allows read and write access to memory starting from address 0x80000000. As a result, we successfully mapped the AP's address space into the CP's address space. As a proof of concept, we patched the first page of the Linux kernel.

```
  unisoc_via_python python3 py_usb.py -r 0x80080000
00000000: 00 40 34 14 00 00 00 00  00 00 08 00 00 00 00 00  .@4.............
00000010: 00 F0 19 00 00 00 00 00  0A 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  70 77 6E 64 00 00 00 00  ........pwnd....
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

At this point, it may seem like the process is complete: we are now able to execute code not only on the modem, but also on the AP. However, the question remains: is there an alternative approach?

In the search for the answer, we decided to investigate the available hardware peripherals on the modem side. The DMA controller was of particular interest to us. Code analysis showed that all DMA controllers are located in the modem's memory area, starting from address 0x20000000. However, any attempt to read memory at these addresses resulted in a DataAbort at the hardware core level. But why?

We analyzed the code responsible for interacting with the DMA controller. Curiously, some sections of this code have no calls to them, as if the developers had forgotten to delete them from the modem OS release version. Using these

fragments of “forgotten” code, we gained access to the DMA controller. It turned out that most of the peripherals are physically powered off. To enable them, it is necessary to write to a special hardware register.

```
LDR     R3, =0x20071034  //unclock DMA
LDR     R2, =0x00000002
STR     R2, [R3]
```

Besides powering it on, the DMA controller also requires a clock signal to operate. This is controlled by another hardware register in the same memory region.

```
LDR     R3, =0x20071060 // enable DMA Clock
LDR     R2, =0x00000001
STR     R2, [R3]
```

Ultimately, we managed to unlock access to the DMA controller, which is unused by the modem’s operating system. We used it, just as we had used MPU tables earlier, to overwrite the first page of the AP’s operating system.

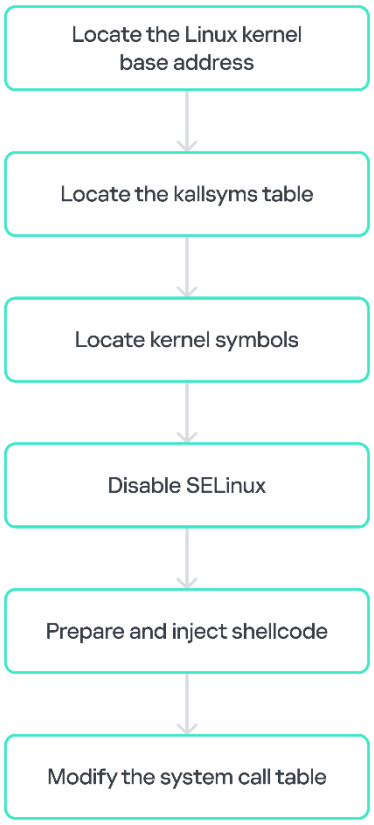
```
ums512_1h10:/ #
ums512_1h10:/ # /data/local/tmp/mem-rw-app -x -m -r 0x80080000 128
Going to operate on address: 80080000 and device /proc/rmem
Length: 128 (0x80) bytes
0000:   00 40 34 14  00 00 00 00  00 00 08 00  00 00 00 00   .@4.............
0010:   00 f0 94 01  00 00 00 00  0a 00 00 00  00 00 00 00   ................
0020:   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0030:   00 00 00 00  00 00 00 00  64 6d 61 20  70 77 6e 64   ........dma pwnd
0040:   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0050:   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
```

Unfortunately, unlike the MPU tables, software updates do not protect against this microarchitectural issue. Clearly, the capabilities it provides can be leveraged to compromise the security of the AP by exploiting a single RCE-level vulnerability in the modem.

Developing an exploit for the AP

After gaining the ability to modify the AP's RAM externally (either by executing code on the modem and reconfiguring its MPU, or by using the DMA peripheral), the next question arises: how can we execute our code on the AP and achieve persistence within its operating system when we have only the ability to read and write its memory?

For demonstration (Proof of Concept), we attempted to install and launch the DOOM game on the AP as a payload. The implementation works as follows: the payload component running on the modem sequentially locates the key structures of the Linux kernel, then injects its own code to perform the necessary actions. The entire process must occur without user interaction and bypass Android's standard security mechanisms.



Algorithm for executing our code on AP side

Step 1: Locate the Linux kernel base address

The first step in the attack is to determine the base address of the Linux kernel loaded into memory. In modern versions of Android, the kernel often uses Address Space Layout Randomization (ASLR), but its base address can be easily determined using known signatures. In our case, this is not even necessary, since the kernel base address is always 0x80080000 (PA) or 0xffffff8008080000 (VA).

Step 2: Locate the kallsyms table

After discovering the kernel's base address, we need to find the kernel symbol table (kallsyms). The table contains the addresses of all exported kernel functions and variables, allowing us to locate the objects required for the attack.

The kallsyms table is usually located in the .rodata section and has a characteristic structure:

- array of symbol addresses;
- array of symbol names;
- index table;
- symbol-type table.

It's worth noting that the array of symbol addresses may use relative addressing instead of absolute addressing to save memory on 64-bit systems, and the symbol name strings may also be compressed. In our case, both of these mechanisms (relative addressing and symbol name compression) were used, complicating the process of locating the table in kernel memory. Nevertheless, the table was constructed using a heuristic algorithm that "assembles" it from data deemed valid and verifies its integrity after assembly.

In our case, we need to find the following symbols, which are critically important for the attack:

- sys_call_table – the system call table;
- call_usermodehelper – a function that launches user mode processes from the kernel;
- selinux_enforcing – the SELinux state flag.

We used the following code to decompress the kernel symbol names.

```rust
fn extract_name<'b>(&self, pos: usize) -> (&'b str, usize) {
    const MAX_NAME_LEN: usize = 256; // Adjust the buffer size as needed
    let name_len = self.names[pos] as usize;
    let mut buffer_index = 0;
    let mut off = pos + 1;
    let mut skipped_first = false;

    unsafe {
        for _ in 0..name_len {
            let token_index = self.names[off] as usize;
            off += 1;
            let token = self.get_token(token_index);

            for &byte in token.as_bytes() {
                if !skipped_first {
                    skipped_first = true;
                } else if buffer_index < MAX_NAME_LEN {
                    BUFFER[buffer_index] = byte;
                    buffer_index += 1;
                }
            }
        }

        let result_str = str::from_utf8(&BUFFER[..buffer_index]).unwrap_or("");
        (result_str, name_len + 1)
    }
}

fn get_token(&self, token_index: usize) -> &str {
    let start = self.token_index[token_index] as usize;
    let end = self.token_table[start..]
        .iter()
        .position(|&c| c == 0)
        .map(|p| start + p)
        .unwrap_or_else(|| self.token_table.len());
    str::from_utf8(&self.token_table[start..end]).unwrap()
}
```

Step 3: Choose a system call to hook

The system call table (sys_call_table) is a key element for the attack. In our system, it is located at address 0x809D2000 (PA) or 0xffffff80089d2000 (VA). The table is an array of pointers to system call handler functions. Each table entry corresponds to a particular system call, identified by a number. These numbers are fixed for each architecture; for ARM64, for example, they can be found on Chromium OS Docs.

We are interested in the getpriority system call with the number 141 (0x8d). The corresponding entry in the table is located at offset 141*8 = 0x468 = 1128 bytes. Thus, the pointer to the getpriority handler is at address 0x809D2468 (PA).

Step 4: Locate the call_usermodehelper function

Once the kernel symbol table (kallsyms) has been found and reconstructed, obtaining the address of the call_usermodehelper function becomes an easy task. In our system, this function is located at 0xffffff80080bfe00. It allows kernel-mode drivers to launch user-mode processes, and this function is what we will use to load, install, and launch the DOOM APK file.

Step 5: Disable SELinux

For the attack to succeed, we need to disable SELinux enforcement temporarily (or permanently); otherwise, it will prevent user-mode processes from being launched via User Mode Helper. To do this, we look up the address of the global variable selinux_enforcing in the kernel symbol table. Setting it to 0 disables SELinux policy enforcement.

Step 6: Find a memory area for code injection

Now we need to find a suitable place in kernel memory to place the shellcode. This should be an unused "hole" in the kernel code section where additional code can be placed without disrupting system operation. In our system, the code section occupies the region from 0x80080800 to 0x809d0000 and is 0x94f800 bytes in size. Using static analysis, we found a sufficiently large free memory region at address 0x809cb000, closer to the end of the kernel code section. Since memory is allocated in pages, free space can often be found at the end of sections, and using it will not affect system functionality in any way.

Step 7: Build and inject shellcode

The task of the shellcode is to execute several commands in the user space. For example, the Activity Manager (am) can be used to launch the installed application: */system/bin/am start -n com.eltechs.originaldoom/.doomDemo.DoomDemo.*

Before this, the shellcode must perform a number of other important actions:

- provide protection against re-execution;
- save the processor context;

- set environment variables required by Package/Activity Manager;
- call the original getpriority handler.

The shellcode is precompiled, position-independent machine code for the ARM64 architecture, at the end of which is a table of pointers to the necessary elements, which is populated on the modem side.

```
printk_address:
    .quad 0xffffff800810a6c0      // printk function
call_umh_address:
    .quad 0xffffff80080bfe00      // call_usermodehelper function
syscall_address:
    .quad 0xffffff80080bbebc      // original getpriority
argv:
    .quad 0xffffff800915df00      // array of arguments
envp:
    .quad 0xffffff800915df40      // environment variables
marker_address:
    .quad 0xffffff800915dff0      // one-time execution marker
```

The code also contains a table of environment variables required for the correct operation of user-space processes such as am (Activity Manager), pm (Package Manager), and others.

```
umh_env_path:

    .asciz

"PATH=/sbin:/system/sbin:/product/bin:/apex/com.android.runtime/bin:/system/bin:/sy
stem/xbin:/odm/bin:/vendor/bin:/vendor/xbin"

umh_env_boot_cp:

    .asciz

"BOOTCLASSPATH=/apex/com.android.runtime/javalib/core-
oj.jar:/apex/com.android.runtime/javalib/core-libart.jar:..."
```

Step 8: Modify the system call table

The final step is to replace the pointer to the getpriority system call handler in sys_call_table with the address of our shellcode, which has already been placed in a free area within the kernel code section. After a successful modification, our code is automatically executed the next time getpriority is called. Since various Android components frequently use this system call, activation occurs within a short period without further intervention.

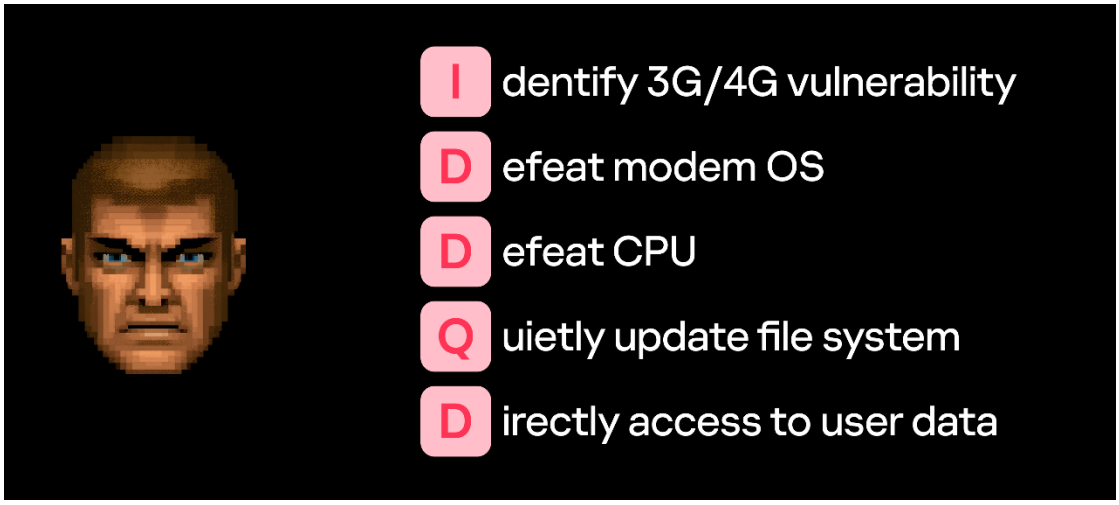
At the end of its operation, the shellcode restores the processor register state and calls the original getpriority system call.

```
    // Restoring registers
    ldp x9, x30, [sp], #16
    ldp x6, x8, [sp], #16
    ldp x4, x5, [sp], #16
    ldp x2, x3, [sp], #16
    ldp x0, x1, [sp], #16
call_original:
    // Calling the original function
    adr x2, syscall_address
    ldr x2, [x2]
    br x2
```

As a result of the entire exploit chain, we gain the ability to execute our own code on the AP.

Now we can confidently say that the head units of certain Chinese vehicles are suitable for playing Doom. The entire process is performed remotely and without any user interaction, successfully bypassing all existing Android and Linux kernel security mechanisms thanks to direct memory access.

Conclusion



Exploiting just one vulnerability on the modem side has provided complete control ("God Mode") over the entire SoC. Importantly, although this vulnerability can be fixed via software, the microarchitectural issues discovered can only be addressed in future batches of the specific SoC. A reasonable question arises: Is this hardware "feature" unique to this particular SoC?

By taking control of the SoC, an attacker not only gains the ability to control the information flow between the device and the outside world but also obtains virtually unlimited access to the most critical components of the end device. If the SoC used in the vehicle's head unit is compromised, it is unlikely that an attacker would stop at installing Doom. They could gain remote access to user data, including intercepting audio through the built-in microphone, or extend the attack to connected mobile devices. Moreover, if a configuration error exists in the onboard CAN bus gateway, an attacker might even be able to impact other vehicle control units remotely[3].

The problem is further aggravated by the fact that, when a serious vulnerability is discovered in a modem, it may take a significant amount of time to update all devices that use that SoC. In some devices, remote update functionality may not be implemented at all. In such cases, installing the update will require additional effort and cost on the part of the end device manufacturer, since every vulnerable SoC would have to be updated manually.

[3] For more information on threats related to remote impact on vehicles, read Modern vehicle cybersecurity trends and "Security researchers are the main factor motivating automakers to invest in protecting their products".

Kaspersky Industrial Control Systems Cyber Emergency Response Team (Kaspersky ICS CERT) is a global Kaspersky project aimed at coordinating the efforts of automation system vendors, industrial facility owners and operators, and IT security researchers to protect industrial enterprises from cyberattacks. Kaspersky ICS CERT devotes its efforts primarily to identifying potential and existing threats that target industrial automation systems and the industrial internet of things.

Kaspersky ICS CERT ics-cert@kaspersky.com