Caliptra Security Assessment

Microsoft

Version 1.2 – October 13, 2023

Prepared By
Andrew Kisliakov
Domen Puncer Kugler
Jeremy Boone
Chris Bury

Prepared For
Eric Eilertson (Microsoft)
Þórður Björnsson (Google)

1 Table of Contents

2 Executive Summary

Synopsis

During August and September of 2023, Microsoft engaged NCC Group to conduct a security assessment of Caliptra[1], a hardware/firmware IP for datacenter-focused server class ASICs. The audit was performed per the requirements outlined in the Open Compute Project's[2] *Security Appraisal Framework & Enablement* (SAFE) program.

Caliptra serves as the internal root-of-trust (iRoT) for both measurement (RTM) and identity of a system-on-chip (SoC). The main use cases for Caliptra are to assure integrity of mutable code, to authorize firmware updates, and to support secure platform configuration and lifecycle state transitions. Following the NIST SP800-193[3] guidelines, Caliptra plays a key role in maintaining resilience of the overall ASIC and the firmware components contained within it. Notably, Caliptra also implements the TCG DICE[4] Protection Environment[5] (DPE) API, enabling other entities within the SoC to leverage the unique device identity for their own security operations.

The security assessment was performed by two (2) consultants over the course of 30 person-days of testing. Two (2) additional consultants provided support in the form of technical oversight and shadowing.

Retesting

During the week of October 9th, NCC Group retested and verified fixes for all reported vulnerabilities. During this time, one new issue (NCC-MSFT283-T3L) was discovered and promptly reported to the Caliptra team. By October 13th, all findings contained in this report were correctly fixed.

Scope

NCC Group's security evaluation of Caliptra spanned the following components:

- **ROM**: The immutable mask ROM, which executes when Caliptra is brought out of reset.
- **First Mutable Code**: Started by the ROM, the FMC is responsible for loading the runtime.
- **Firmware**: The runtime firmware which provides Caliptra's services to the SoC.

Microsoft furnished NCC Group with several testing objectives and focus areas for this project. These requirements are, for the most part, related to upholding the desired security properties of the DICE Protection Environment, including protection of its sensitive assets, such as the Unique Device Secret (UDS) and Composite Device Identifier (CDI).

- Ensure that Caliptra's firmware loading and validation process cannot be bypassed.
- Prevent attacks that undermine DICE initialization and external firmware measurement.
- Ensure that measurements cannot be silently dropped or excluded from DPE derivations.
- Review DPE signing for side-channel information leakage, impacting the UDS/CDI's.
- Determine whether an attacker can corrupt the DPE context tree structure.
- Ensure that cryptographic material is cleared from memory after use.
- Under debug, DPE certificates should not chain to vendor-signed DeviceID certificates.
- Assess the effectiveness of exploit mitigation technologies.
- Assess the soundness of the fault injection countermeasures.

1. The CHIPS Alliance's Caliptra Project
2. The Open Compute Project
3. NIST Special Publication 800-193 - Platform Firmware Resiliency Guidelines
4. Trusted Computing Group - DICE Specification
5. Trusted Computing Group - DICE Protection Environment Specification

Testing was performed on the open-source code hosted in GitHub. Although not explicitly in scope, when necessary, NCC Group referred to the Caliptra RTL to better understand the underlying hardware logic (i.e., the mailbox state machine, and ECC signature verification). A full listing of resources can be found in the Provided Materials section.

Limitations

Only one minor issue arose. That is, the latest revision of DPE specification is currently in draft and unfortunately is private to Trusted Computing Group (TCG) members. As a result, NCC Group was forced to audit the Caliptra DPE Profile source code while using outdated information in an older revision of the specification.

Beyond this, NCC Group's review of Caliptra was not inhibited by any significant factors which delayed progress or prevented deep analysis.

Key Findings

The assessment uncovered several noteworthy security flaws related to the DICE Protection Environment that undermine its properties of integrity, confidentiality and availability:

- Changes in one context could impact the measurements included in operations of other contexts (NCC-MSFT283-6BV)
- A context tree could be corrupted in a way that affects parent-child relations on newly created contexts (NCC-MSFT283-KML)
- Sensitive context handles could be exposed to an attacker via a timing side-channel (NCC-MSFT283-PTP)
- The context handle array could be filled by malicious or genuine operations, leading to a denial of service (NCC-MSFT283-29Q)

A high risk vulnerability was also found in the ROM code that verifies the firmware images. A malicious SoC could interfere with the SHA-512 Accelerator causing a clean firmware image to be verified, but a malicious firmware image to be loaded and executed (NCC-MSFT283-YMG). A related weakness elsewhere in the code could lead to premature release of the SoC's SHA Accelerator lock (NCC-MSFT283-3QD), opening up another avenue for these types of race conditions to occur.

Many reported findings were determined to convey a low overall impact, but it is important to recognize that even low risk issues can be exploited when combined with other issues as part of a wider attack. However, in practice, many of these low risk findings are simply vectors for denial of service attacks (e.g., NCC-MSFT283-4CR). Depending on Caliptra's threat model, these denial-of-service concerns may be more or less problematic.

Finally, several findings were reported only for informational purposes, out of an abundance of caution. These informational issues do not describe current vulnerabilities, but rather serve to highlight minor issues such as weak API designs that could lead to misuse, or discrepencies between Caliptra's specification and implementation.

Positive Observations

1. Use of Rust

In NCC Group's experience, memory safety violations are the primary source of vulnerabilities that afflict embedded systems. Memory safety is especially important for root-of-trust implementations[6], which must have an elevated security posture due to their responsibility as the trust anchor for the entire platform. Without memory safety protections, the overall security of an embedded system is usually only one or two vulnerabilities away from compromise. NCC Group strongly believes that all new firmware projects should be written in memory safe languages, as the Caliptra authors have done here.

Caliptra's Rust firmware contains hundreds of unsafe blocks. Although NCC Group made an effort to review each unsafe block, there were simply too many to cover in the time allotted for this engagement. Instead, we carefully considered the unsafe blocks that were encountered while addressing the other in-scope tasks. Specifically, we prioritized the unsafe blocks that were reachable from Caliptra's external attack surfaces, such as the mailbox interface and the firmware loading flows. The majority of the unsafe blocks were observed to use constant or tightly constrained values, such as referencing fixed register addresses. These were judged to expose no risk to memory safety as they do not process untrusted attacker-controlled inputs.

NCC Group did not find any critical memory safety violations which would violate the integrity or confidentiality requirements of the Caliptra platform. Only a single low-impact memory corruption concern (NCC-MSFT283-4DN) was discovered. Furthermore, all NMI and exceptions are handled as fatal failures, as is Rust's panic, which means that out-of-bounds array accesses (for example) would also trigger a fatal error.

Overall, Caliptra's posture on memory safety appears to be quite strong. We applaud the decision to implement Caliptra in Rust.

2. Mailbox Protocol

Caliptra implements two mailbox interfaces that are exposed to SoC agents – a command mailbox, and a SHA Accelerator mailbox. NCC Group discovered race conditions that impact the SHA mailbox, as described in the Key Findings above. However, the command mailbox was unique in that the messaging protocol's state machine is implemented in hardware. The hardware logic enforces several properties:

1. Via a locking mechanism, ensure that only a single entity can interact with the mailbox,
2. Strictly enforce the correct ordering of API register writes,
3. Copy the command payloads out of the mailbox registers to an SRAM buffer,
4. Ensure that the 'data length' register is not larger than the SRAM buffer size.

If any of these properties are violated, the mailbox state machine will enter an error state and will throw an interrupt, safely halting processing of all mailbox operations. This hardware logic is an effective mitigation against TOCTOU vulnerabilities.

3. Elliptic Curve RTL

Although not in scope, NCC Group briefly audited the hardware implementation of the elliptic curve cryptographic primitives. This effort was taken because ECDSA is crucial to the overall security posture of Caliptra due to its usage in the secure boot mechanism. The RTL was analyzed to ensure that common cryptographic pitfalls were avoided, such as validating that the point coordinates are less than the field modulus and on the curve, and that the point is not at infinity and in the correct subgroup.

6. NCC Group - LeaPFRogging PFR Implementations

Largely, the Verilog that implemented these assertions was observed here and here. It was found to be free of the aforementioned flaws.

4. Caliptra Memory Protections

NCC Group reviewed the memory access protections offered by the Chips Alliance's RISC-V VeeR EL2[7] core. The goal of this analysis was to determine whether the data-only memory regions were executable, or whether the code-only memory regions were writable. If either of these properties were violated, then some classes of memory safety exploits would be made possible.

Caliptra splits its memory into two areas – the Instruction and Data Close Coupled Memories (ICCM and DCCM) – and our analysis for each is summarized below.

In reviewing the memory layout of Caliptra which is created by the image bundle generator, it was noted that the **ICCM** memory region contains a mixture of both code and data sections (a concatenation of `.text`, `.rodata` and initial `.data`). However, near the end of ROM execution, the ICCM region is locked to prevent further writes. Although this locking behavior doesn't prevent execution of data in the ICCM, it does prevent a potential attacker from writing data into the ICCM. NCC Group believes that it is unlikely an attacker would have an opportunity to stage a payload in ICCM prior to it being locked, and so it will be very difficult to achieve code execution in ICCM.

Caliptra's riscv32imc core does not have a memory management unit (MMU), but it does have a rudimentary memory protection unit (MPU). The MPU is used to protect the **DCCM** region, which appears to only contain the `.stack` sections for the ROM, FMC, Runtime firmware, as well as the exception and NMI handlers. The `.bss` and `.data` sections are always empty. Furthermore, the specification[8] explains further constraints which might make it difficult for an attacker to execute a staged payload in DCCM:

> *An instruction fetch to a non-ICCM region must fall within the address range of at least one instruction access window for the access to be forwarded to the IFU bus interface*

Overall, our analysis determined that the property of W^X is adequately satisfied. As a result of this, the most likely avenue for an attacker to achieve arbitrary code execution would be through a ROP-style of exploit. However, Caliptra aims to mitigate such exploits through its Control Flow Integrity (CFI) mechanism.

7. Open source RTL for the CHIPS Alliance's VeeR EL2 core
8. RISC-V VeeR EL2 Programmer's Reference Manual, Section 2.6 "Memory Protection"

Strategic Recommendations

1. Implement the BMI Profile

Caliptra supports two methods of integration with a SoC which dictate how Caliptra retrieves its runtime firmware while booting: The Boot Media Integrated (BMI) Profile and the Boot Media Dependent (BMD) Profile[9]. In past versions of Caliptra's specification, these were referred to as the Active Profile and the Passive Profile.

- **BMI Profile**: Caliptra's ROM loads its own firmware from persistent flash storage.
- **BMD Profile**: Caliptra's ROM exposes a minimal mailbox interface, allowing a SoC agent to load Caliptra's runtime firmware. In this case, Caliptra is still responsible for verifying the cryptographic integrity of this firmware image.

At the moment, Caliptra supports only the BMD Profile which introduces a minor dilemma – The dimensions of the platform's Trusted Computing Base (TCB) is increased in devices that leverage the BMD Profile because the trusted envelope must be expanded to encompass the SoC agent that loads Caliptra's firmware.

In other words, the BMD Profile creates a chicken-egg problem wherein Caliptra cannot act as the root-of-measurement (RTM) for the SoC as a whole. Caliptra has no means to verifiably measure the firmware of the SoC agent that loaded Caliptra's firmware. When using the BMD Profile, the actual *root* of the RTM is not Caliptra, but instead, the root is the SoC agent which loaded Caliptra's FMC and runtime firmware.

NCC Group encourages the Caliptra Working Groups to finish development of the BMI Profile, thus enabling Caliptra to load its own firmware from external flash without needing to coordinate with glue logic elsewhere in the SoC.

2. Enhance Fault Injection Countermeasures

Caliptra's only form of defense against fault injection attacks is its Control Flow Integrity solution, which protects the forward-edge of the call-graph. The CFI library[10] manipulates the program's AST, and instruments function calls to insert several operations:

1. Read a hardware counter
2. Delay for a random duration
3. Call the original function (which is amended to increment the CFI counter)
4. Delay for a random duration
5. Decrement the counter
6. Compare the original counter value to the decremented value
7. Panic if the counter values mismatch

This CFI solution is primarily aimed at mitigating fault injection attacks that target control-flow influencing instructions (i.e., branch, or jump). That is, the CFI mechanism will panic if an attacker introduces a glitch which causes an instruction-skipping fault and prevents a function from being invoked. In practice, this type of mitigation is useful for protecting security-critical functions, such as those involved in secure boot signature validation, ensuring that an attacker can never "step over" those functions with a carefully timed glitch.

Furthermore, the inserted random delays are useful to reduce the reliability of any glitching attempts. These delays make it difficult for an attacker to accurately predict when to inject their fault after an externally-observable event. By adding random delays, Caliptra is somewhat able to mask these trigger events that an attacker relies upon to infer the correct

9. Caliptra Boot Profile Documentation
10. Caliptra's CFI Library

moment to inject the fault. Example trigger conditions may include power analysis, observing bus traffic, or a fixed time offset after a command is sent to a mailbox.

However, CFI is not a perfect remedy to defend all types of fault injection, and there are several limitations of Caliptra's implementation.

Limitation 1: CFI Coverage is Incomplete

Most software-based fault injection countermeasures require dozens or hundreds of small and similar changes to be sprinkled throughout the entire code base. The manual process of applying these code changes can be fragile and prone to human error. The same concern is also true for Caliptra.

First, Caliptra's CFI solution only protects functions which are specifically annotated with the `cfi_impl_fn` or `cfi_mod_fn` attributes. Because the developer has to make a conscious decision to protect a function, this can lead to oversights where security-critical operations are not guarded by the CFI mechanism.

Second, the CFI attributes only appear to be used by Caliptra's ROM, and do not appear in the FMC or Runtime Firmware. Even within the ROM, CFI coverage is incomplete. Specifically, library functions and drivers used by the ROM do not make use of CFI.

Consequently, Caliptra's CFI implementation contained gaps where critical functions were left unprotected, as discussed in NCC-MSFT283-BKC. Furthermore, after carefully analyzing the CFI library, NCC Group discovered that the CFI primitives could be further enhanced as described in NCC-MSFT283-T3L. Finally, after delivering the first draft of this report, Google opened two new pull requests[11] [12] for adding glitch defenses to other critical functions.

All together, these findings serve to highlight our assertion that applying software-based fault injection countermeasures to a large firmware project is akin to playing whack-a-mole. It is all too easy to overlook protections on critical code paths, and the CFI primitives themselves may be glitchable.

Limitation 2: Memory Load/Store Not Protected

Although CFI aims to protect branch instructions, it does not protect memory load or store instructions. This shines a spotlight on another gap in Caliptra's glitching defenses.

A common target for fault injection attacks is to introduce a glitch when security-critical values are read from persistent storage mediums. Examples of this are flash, fuses, or registers, which may store important flags such as whether debug is enabled, or whether a key slot is valid. If an attacker can accurately inject a fault while the firmware is reading a register or fuse, they may be able to flip one or more bits in the returned result[13].

The typical defense against glitches that target memory load instructions is to use redundant reads which are interspersed by random delays. Each time the value is read, it should be compared to ensure it matches the earlier reads.

Another common defense is to store critical flags using multi-bit encodings, such that no single bit flips can result in a state transition. This mitigation relies heavily on the fact that glitching outcomes are often unpredictable, and an attacker cannot control which bit is flipped in the returned result. If state transitions require multi-bit encodings, then the attacker's chances of success are reduced.

After delivering an initial draft of this report, these suggestions were taken as hardening opportunities by the Caliptra team. For example, new GitHub issues were opened for (1)

11. Caliptra SW Issue #921 - "Triple call CFI Counter reset when seeding the prng"
12. Caliptra SW Issue #922 - "Mitigate glitching of calls within Trng generate"
13. NCC Group - An Introduction To Fault Injection

performing strict data verification for the TRNG peripheral's MMIO[14], and (2) enhacing the encoding of Caliptra's security state[15].

Hardware-Based Countermeasures

Ultimately, many experts view software-based fault injection countermeasures as being of limited value, because they do not completely eliminate the problem. Instead, software-based defenses merely reduce an attacker's success rate. In many physical attack scenarios, the threat actor will be in possession of the victim device for an extended period of time, and will be able to repeatedly attempt glitching until successful.

The most effective defenses against fault injection attacks are those that are hardware-based. These may be fast-reacting voltage/current sensors within the silicon (whose purpose is to detect anomalies outside the intended electrical operational parameters), or a carefully calibrated Tunable Replica Circuit (whose purpose is to detect circuit timing violations), or shadow stacks (whose purpose is to protect the reverse edge of the control flow graphs). There are several other solutions in this space, and NCC Group encourages Caliptra to adopt one or more in a future hardware revision.

3. Remove SoC Access to SHA-512 Accelerator

The locking mechanism used by the SHA-512 Accelerator to prevent concurrent usage was found to be incomplete. As described in NCC-MSFT283-YMG, a vulnerability that could lead to execution of unsigned code was exposed through a non-atomic check used to determine that the lock had been acquired by Caliptra, together with the fact that Accelerator registers including the digest register were readable by Caliptra while locked by the SoC. The latter factor appeared to be necessitated by a design requirement to satisfy an unrelated future use case which had been communicated to NCC Group.

NCC Group recommends that external access to the SHA-512 Accelerator be removed. This would remove the possibilities for any malicious interactions from the SoC. The above-mentioned use case could be replaced by the SoC providing the digest through a Mailbox command for example. Alternative means to reduce the chance of potential abuse might include removing its use from the image verification process or removing Caliptra's access to it for any purposes other than reading the digest register.

4. Clarify Warm Reset Handling

The term "non-orderly" is sometimes used to distinguish between expected resets and unexpected ones. Embedded systems must be cautious when recovering from a non-orderly reset because the firmware state may be corrupted. Vigilance is needed to verify all data that was preserved across the reset event.

A common attack vector for embedded systems is for an adversary to trigger a non-orderly reset in the middle of a sequence of sensitive write operations. Such a well-timed reset would prevent some or all writes from completing. For example, this could lead to (1) a denial of service condition, (2) undermine expectations of atomicity of persistent data, or (3) prevent incrementing of monotonic counters.

Failure to properly handle non-orderly resets has led to serious vulnerabilities in the past, such as a bypass of the Trusted Platform Module's Dictionary Attack (DA) lockout mechanism[16].

14. Caliptra SW Issue #920 - "Triple-verify TRNG MMIO read at CFI initialization in rom_entry"
15. Caliptra RTL Issue #243 - "Enhancement: fault-injection-aware encoding for security state"
16. An example vulnerability arising from non-orderly shutdown in ms-tpm-20-ref

Regarding Caliptra, there is a known outstanding issue posted on GitHub[17] related to warm resets. This issue has not seen much progress towards resolution. In Caliptra, a warm reset may occur asynchronously, and it is interesting from a threat modeling perspective because registers and memories may stay intact across the reset. This can put the silicon in a different state than might be expected – for example, some Data Vault entries get unlocked for writes, but their contents remain intact.

Currently, warm reset handling depends on the `ColdResetComplete` flag to be set and the ROM just transfers control to FMC. While this excludes most of the cold reset flow (there is still a short window after the complete flag is set and before the `FirmwareHandoffTable` is fully populated), it does not exclude all of the ROM code. Consequently, a malicious SoC agent could request a firmware update and then trigger a warm reset somewhere in the middle of update reset handling. This extends the warm reset “entry points” to ROM’s update reset code, in addition to the more obvious FMC or Runtime.

In Caliptra, warm resets should have a clearly defined handling, which might include putting registers, vaults and peripherals in a predefined “non-orderly reset” state, and should most likely restrict the functionality of FMC and Runtime.

17. Caliptra SW Issue #167 - Warm Reset can lead to undefined state

3 Dashboard

Target Data

Name	Caliptra
Type	Firmware
Platforms	Rust on RISC-V
Environment	Whitebox

Engagement Data

Type	Firmware Review
Method	Code-assisted
Dates	2023-08-16 to 2023-09-14
Consultants	2
Level of Effort	30 person-days

Targets

ROM	The immutable mask ROM
FMC	The first mutable code loaded by the ROM
Runtime Firmware	The firmware loaded by the FMC that offers Caliptra services to the SoC

Finding Breakdown

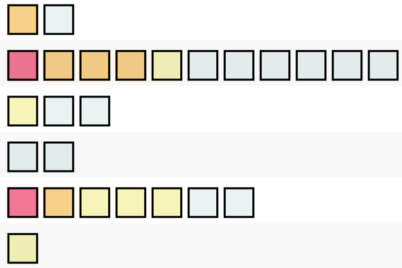
Critical issues	0
High issues	2
Medium issues	5
Low issues	6
Informational issues	13
Total issues	**26**

Category Breakdown

Configuration	1
Cryptography	2
Data Exposure	4
Data Validation	2
Denial of Service	4
Other	5
Patching	1
Policy Violation	1
Security Improvement Opportunity	4
Timing	2

Component Breakdown

General	2	
DPE	11	
Drivers	3	
FMC	2	
ROM	7	
libcaliptra	1	



Critical High Medium Low Informational

4 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.

General

Title	Status	ID	Risk
CFI Assert Macros Offer Limited Glitching Protection	Fixed	T3L	Medium
Outdated Dependencies	Fixed	BT3	Info

DPE

Title	Status	ID	Risk
Changes in Context Tree Affecting Behaviour in Other Branches	Fixed	6BV	High
Timing Side-Channel Exposes Context Handles	Fixed	PTP	Medium
DestroyCtx Command Can Corrupt Context Tree	Fixed	KML	Medium
Multiple Ways to Exhaust DPE Context Handles	Fixed	29Q	Medium
Premature Context State Modification in DeriveChild	Fixed	6Y9	Low
ChildToRootIter Infinite Loop	Fixed	VF7	Info
Test Code May Panic on Some Inputs	Fixed	97Q	Info
Chunk Size and Certificate Size Misuse in GetCertificateChain Command	Fixed	VKD	Info
Context Handles Not Rotated on Error	Risk Accepted	DBR	Info
CryptoBuf Can Be Partially Initialized	Fixed	32H	Info
DeriveChild Permits Context Handle to Coexist With the Default Context	Fixed	DDK	Info

Drivers

Title	Status	ID	Risk
MailboxSendTxn drop() Handling Not Exhaustive	Fixed	4CR	Low
Random Number Generation Iterator Potentially Returning Non-Random Values	Fixed	2QM	Info
LMS Verifier Permitted Invalid q Value	Fixed	NTF	Info

FMC

Title	Status	ID	Risk
Comment and Code Mismatch in derive_cdi	Fixed	GGG	Info
Insufficient Validation of Memory Addresses	Fixed	QBA	Info

ROM

Title	Status	ID	Risk
TOCTOU in SHA-512 Accelerator Lock Acquisition	Fixed	YMG	High
Premature Release of SHA-512 Accelerator Lock	Fixed	3QD	Medium
slice::fill(0) Does Not Always Zero Memory	Fixed	962	Low
Buffer Overflow in PCR Logging	Fixed	4DN	Low

Title	Status	ID	Risk
Critical Functions Not CFI Protected	Fixed	BKC	Low
Memory Not Cleared During Error Conditions	Fixed	42W	Info
ROM Integrity Test Does Not Cover .data Section	Fixed	BQM	Info

libcaliptra

Title	Status	ID	Risk
TOCTOU Condition in File Read Leading to Uninitialised Memory Buffer	Fixed	VH3	Low

5 Finding Details – General

Medium

CFI Assert Macros Offer Limited Glitching Protection

Overall Risk	Medium	**Finding ID**	NCC-MSFT283-T3L
Impact	Medium	**Component**	General
Exploitability	Low	**Category**	Security Improvement Opportunity
		Status	Fixed

Impact

Within Caliptra's Control Flow Integrity library, the assert implementation offers limited protection against fault injection attacks.

Description

CFI assertions are implemented as macros and are shown below.

```
macro_rules! cfi_assert_macro {
    ($name: ident, $op: tt, $trait1: path, $trait2: path, $panic_info: ident) => {
        /// CFI Binary Condition Assertion
        ///
        /// # Arguments
        ///
        /// `a` - Left hand side
        /// `b` - Right hand side
        #[inline(never)]
        #[allow(unused)]
        pub fn $name<T>(lhs: T, rhs: T)
        where
           T: $trait1 + $trait2,
        {
           if cfg!(feature = "cfi") {
               CfiCounter::delay();
               if !(lhs $op rhs) {
                   cfi_panic(CfiPanicInfo::$panic_info);
               }
           } else {
               lhs $op rhs;
           }
        }
    };
}

cfi_assert_macro!(cfi_assert_eq, ==, Eq, PartialEq, AssertEqFail);
cfi_assert_macro!(cfi_assert_ne, !=, Eq, PartialEq, AssertNeFail);
cfi_assert_macro!(cfi_assert_gt, >, Ord, PartialOrd, AssertGtFail);
cfi_assert_macro!(cfi_assert_lt, <, Ord, PartialOrd, AssertLtFail);
cfi_assert_macro!(cfi_assert_ge, >=, Ord, PartialOrd, AssertGeFail);
cfi_assert_macro!(cfi_assert_le, <=, Ord, PartialOrd, AssertLeFail);

#[macro_export]
macro_rules! cfi_assert {
```

```
    ($cond: expr) => {
        cfi_assert_eq($cond, true)
    };
}
```

Figure 1: caliptra-sw/cfi/lib/src/cfi.rs:143-181

Skimming the above code, one might expect that a delay is first executed followed by a comparison of `lhs` and `rhs` expressions. However, the macros generate functions, so `lhs` and `rhs` are already evaluated after macro expansion.

We demonstrate the deficit of assertions on the following short function `WarmResetFlow::run()`. Similar constructs can be seen in image validation code that uses `svn_check_required()`. Highlighted are the boot status reads for which the intention is to protect them from glitching attacks where in invalid value might be read or a comparison skipped.

```
    pub fn run(env: &mut RomEnv) -> CaliptraResult<()> {
        cprintln!("[warm-reset] ++");

        // Check if previous Cold-Reset was successful.
        if cfi_launder(env.data_vault.rom_cold_boot_status()) != ColdResetComplete.into() {
           cprintln!("[warm-reset] Previous Cold-Reset was not successful.");
           return Err(CaliptraError::ROM_WARM_RESET_UNSUCCESSFUL_PREVIOUS_COLD_RESET);
        } else {
           cfi_assert!(env.data_vault.rom_cold_boot_status() == ColdResetComplete.into());
        }

        cprintln!("[warm-reset] --");

        Ok(())
    }
```

Figure 2: caliptra-sw/rom/dev/src/flow/warm_reset.rs:31-45

For the RISC-V target, the above Rust code compiles to following Assembly. Some annotations were added by NCC Group to make it easier to follow.

```
00005408 <caliptra_rom::flow::warm_reset::WarmResetFlow::run>:
    5408:       1101                    add     sp,sp,-32
    540a:       ce06                    sw      ra,28(sp)
    540c:       cc22                    sw      s0,24(sp)
    540e:       ca26                    sw      s1,20(sp)
    5410:       c84a                    sw      s2,16(sp)
    5412:       50000537                lui     a0,0x50000
    5416:       3e452903                lw      s2,996(a0) # 500003e4 <DATA_ORG+0x3e4>
    541a:       3e852483                lw      s1,1000(a0)
    541e:       234d                    jal     59c0
    ↳<caliptra_cfi_lib::cfi_counter::CfiCounter::delay>
    5420:       29fd                    jal     591e
    ↳<caliptra_cfi_lib::cfi_counter::CfiCounter::increment>
    5422:       00008537                lui     a0,0x8
    5426:       36450513                add     a0,a0,868 # 8364
    ↳<.Lanon.a5906495874ff3d10b5fd20dd6ecf31b.25+0x10>
    //  8364 5b776172 6d2d7265 7365745d 202b2b0a  [warm-reset] ++.
    542a:       45c1                    li      a1,16
    542c:       0ab000ef                jal     5cd6 <<caliptra_drivers::printer::Printer as
    ↳ufmt_write::uWrite>::write_str>  // cprintln!("[warm-reset] ++");
    5430:       1001c537                lui     a0,0x1001c
```

```
    5434:       4a452583                lw      a1,1188(a0) # 1001c4a4
    ↳ <DCCM_SIZE+0xfffc4a4>  // env.data_vault.rom_cold_boot_status()
    ↳ (sticky_lockable_scratch_reg == 0x1001c4a0, ColdResetEntry4::RomColdBootStatus == 1)
    5438:       c62e                    sw      a1,12(sp)
    543a:       006c                    add     a1,sp,12
    543c:       45b2                    lw      a1,12(sp)
    543e:       14000613                li      a2,320          // ColdResetComplete
    5442:       02c59363                bne     a1,a2,5468 <.LBB29_2>
    5446:       4a452503                lw      a0,1188(a0) //
    ↳ env.data_vault.rom_cold_boot_status()
    544a:       ec050513                add     a0,a0,-320
    544e:       00153513                seqz    a0,a0   // a0 if a0 zero;
    ↳ rom_cold_boot_status() == ColdResetComplete
    5452:       e07fa0ef                jal     258 <caliptra_cfi_lib::cfi::cfi_assert_eq>
    5456:       00008537                lui     a0,0x8
    545a:       37450513                add     a0,a0,884 # 8374
    ↳ <.Lanon.a5906495874ff3d10b5fd20dd6ecf31b.25+0x20>
    //  8374 5b776172 6d2d7265 7365745d 202d2d0a  [warm-reset] --.
    545e:       45c1                    li      a1,16
    5460:       077000ef                jal     5cd6 <<caliptra_drivers::printer::Printer as
    ↳ ufmt_write::uWrite>::write_str>
    5464:       4401                    li      s0,0
    5466:       a821                    j       547e <.LBB29_3>

00005468 <.LBB29_2>:  // if (rom_cold_boot_status() != ColdResetComplete)
    5468:       00009537                lui     a0,0x9
    546c:       9b450513                add     a0,a0,-1612 # 89b4
    ↳ <.Lanon.a5906495874ff3d10b5fd20dd6ecf31b.92>
    //  89b4 5b776172 6d2d7265 7365745d 20507265  [warm-reset] Pre
    //  89c4 76696f75 7320436f 6c642d52 65736574  vious Cold-Reset
    //  89d4 20776173 206e6f74 20737563 63657373   was not success
    //  89e4 66756c2e 0a5b7374 6174655d 204c6f63  ful..[state] Loc
    5470:       03500593                li      a1,53
    5474:       063000ef                jal     5cd6 <<caliptra_drivers::printer::Printer as
    ↳ ufmt_write::uWrite>::write_str>
    5478:       01040437                lui     s0,0x1040
    547c:       0441                    add     s0,s0,16 # 1040010 <DCCM_SIZE+0x1020010>

0000547e <.LBB29_3>:
    547e:       2389                    jal     59c0
    ↳ <caliptra_cfi_lib::cfi_counter::CfiCounter::delay>
    5480:       29d9                    jal     5956
    ↳ <caliptra_cfi_lib::cfi_counter::CfiCounter::decrement>
    5482:       862a                    mv      a2,a0
    5484:       86ae                    mv      a3,a1
    5486:       854a                    mv      a0,s2
    5488:       85a6                    mv      a1,s1
    548a:       2331                    jal     5996
    ↳ <caliptra_cfi_lib::cfi_counter::CfiCounter::assert_eq>
    548c:       8522                    mv      a0,s0
    548e:       40f2                    lw      ra,28(sp)
    5490:       4462                    lw      s0,24(sp)
    5492:       44d2                    lw      s1,20(sp)
    5494:       4942                    lw      s2,16(sp)
    5496:       6105                    add     sp,sp,32
    5498:       8082                    ret
```

In the highlighted assembly code above it can be seen there were no delays added, meaning that if one operation (a register read, or comparison) can be glitched correctly, so can the duplicate that follows, since it is a deterministic number of cycles away. Additionally, the call to `cfi_assert_eq` could be glitched as well.

The `cfi_assert_eq` does contain a delay, but the effect of that is only to make glitching to skip a call to `cfi_panic` harder. This offers a very weak protection, since attackers will have plenty of suitable glitching targets before.

Recommendation

The CFI assert API should be refactored such that:

1. The arguments are evaluated at least twice.
2. A random-duration delay is inserted between the evaluations.

Retest Results

2023-10-13 – Fixed

Pull request #957 fixes this finding by inlining the assert functions (eliminating the function call which could be glitched) and adding one more delay and comparison.

Info

Outdated Dependencies

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-BT3
Impact	None	**Component**	General
Exploitability	None	**Category**	Patching
		Status	Fixed

Impact

Outdated dependencies with known vulnerabilities have the potential to introduce vulnerabilities into the product. Although a small number of vulnerable dependencies were found within the Caliptra codebase, these were determined to pose no risk because they were not included within the ROM, FMC and Runtime built images. Furthermore, the use of the vulnerable libraries within build and test code did not exercise the vulnerable portions of the flagged dependencies. As a result, this finding is provided for informational purposes only.

Description

Several outdated dependencies with known vulnerabilities were detected.

atty 0.2.14 (Informational)

This version was vulnerable to GHSA-g98v-hv3f-hcfr[18]. Because this vulnerability affected only the Windows version of the library, it is not considered to be relevant to the current implementation. However, it should be noted that this library is unmaintained, with no expected fix for the current vulnerability or any other currently unknown vulnerabilities. Because this library was only referenced via other dependencies as a build dependency, it is not expected that the final built product could incorporate the vulnerability in any way.

openssl 0.10.48 (Informational)

This version was vulnerable to GHSA-xcf7-rvmh-g6q4[19].

The following `cargo tree` output describes the points where this dependency was referenced as a dependency of the built product:

```
$ cargo tree -i openssl -e normal
openssl v0.10.48
├── caliptra-image-app v0.5.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/image/
↳ app)
├── caliptra-image-openssl v0.1.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/
↳ image/openssl)
│   ├── caliptra-builder v0.1.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/
↳ builder)
│   │   └── caliptra-size-history v0.1.0 (/home/consultant/projects/2023-Caliptra/code/
↳ caliptra-sw/ci-tools/size-history)
│   └── caliptra-image-app v0.5.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/
↳ image/app)
├── caliptra-test v0.1.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/test)
└── vector_gen v0.1.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/drivers/test-
↳ fw/scripts/vector_gen)
```

18. GitHub Advisory Database: GHSA-g98v-hv3f-hcfr
19. GitHub Advisory Database: GHSA-xcf7-rvmh-g6q4

This dependency tree demonstrates that the library was not included within the ROM, FMC or Runtime images. It was included as a “build” and “dev” dependency, meaning that it was used in build- and test-specific code only.

The openssl library was included within the `caliptra-image-app` application, which was used to build and sign image bundle files containing FMC and Runtime code. However, the GHSA-xcf7-rvmh-g6q4 vulnerability was deemed to be not applicable in this use case.

time 0.1.45 (Informational)

This version was vulnerable to CVE-2020-26235[20].

The following `cargo tree` output describes the points where this dependency was referenced as a dependency of the built product:

```
$ cargo tree -i time@0.1.45 -e normal
time v0.1.45
└── chrono v0.4.24
    ├── asn1 v0.13.0
    │   └── caliptra-test v0.1.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/
    ↪ test)
    └── caliptra-image-app v0.5.0 (/home/consultant/projects/2023-Caliptra/code/caliptra-sw/
    ↪ image/app)
```

This dependency tree similarly demonstrates that the library was not included within the ROM, FMC or Runtime images, but used within the `caliptra-image-app` application which used to build and sign image bundle files. Because the vulnerability itself was applicable only to multithreaded environments, it was also deemed not to be vulnerable in this use case.

Recommendation

Ensure a regular process of patching out-of-date dependencies to ensure that known vulnerabilities are not introduced into the product.

Continue to minimise the use of third-party dependencies within built ROM, FMC and Runtime images to minimise the possibility that

Location

- caliptra-sw/cargo.lock

Retest Results

2023-10-13 – Fixed

Pull requests #915 and #959 fixed this finding.

20. National Vulnerability Database: CVE-2020-26235

6 Finding Details – DPE

High

Changes in Context Tree Affecting Behaviour in Other Branches

Overall Risk	High	**Finding ID**	NCC-MSFT283-6BV
Impact	High	**Component**	DPE
Exploitability	Medium	**Category**	Other
		Status	Fixed

Impact

The `DeriveChild` DPE command could be invoked in a manner which modified a parent context in a way that might impact the derived key that is used to compute signatures generated using contexts within other branches of a context tree.

By modifying a parent context in this way, an attacker could cause the result of a `Sign` command to be computed using a different derived key pair to that used in a preceding `CertifyKey` command executed within the same context. Within an attestation flow, this could result in the certificate chain returned by the attestation response containing a different public key to that used to sign the attestation challenge.

If the context tree contained derived contexts from different localities, the changes could also affect contexts across different localities.

Description

The implementation of the `DeriveChild` command contained two statements as follows.

```
dpe.contexts[parent_idx].uses_internal_input_info = self.uses_internal_info_input().into();
dpe.contexts[parent_idx].uses_internal_input_dice = self.uses_internal_dice_input().into();
```

Figure 3: caliptra-dpe/dpe/src/commands/derive_child.rs:123-124

These statements set the parent context's `uses_internal_input_info` and `uses_internal_input_dice` flags to what had been specified within the current request.

When calculating a measurement hash, the `DpeInstance::compute_measurement_hash` function iterated from the selected context through its ancestors using the following code:

```
// Hash each node.
for status in ChildToRootIter::new(start_idx, &self.contexts) {
    let context = status?;

    hasher
        .update(context.tci.as_bytes())
        .map_err(|_| DpeErrorCode::HashError)?;

    // Check if any context uses internal inputs
    uses_internal_input_info =
        uses_internal_input_info || context.uses_internal_input_info();
    uses_internal_input_dice =
        uses_internal_input_dice || context.uses_internal_input_dice();
}
```

Figure 4: caliptra-dpe/dpe/src/dpe_instance.rs:351-364

The values of the `uses_internal_input_info` and `uses_internal_input_dice` variables would be set to `true` if at least one of the contexts had the necessary flag set. While this appears to be the true intent of the code, the block described earlier appears to be unnecessary and could lead to the behaviour of certain branches of the context tree changing in response to the addition of new branches, as illustrated in the following example.

1. Initialize a new context C0 with `uses_internal_input_info` set to `false`.

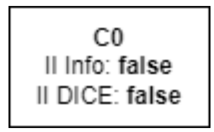


Figure 5: State of Context Tree After Initialising C0

2. From C0, derive a child context C1 with `uses_internal_input_info` set to `false`.
3. Calculate a measurement hash using context C1, which will not include internal input info.

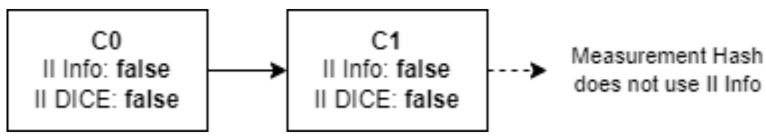


Figure 6: State of Context Tree after Inserting C1

4. From C0, derive a child context C2 with `uses_internal_input_info` set to `true`. The `uses_internal_input_info` flag in C0 will change to `true`.
5. Calculate a measurement hash using context C1, which now will include the internal input info because the C0 flag is `true`.

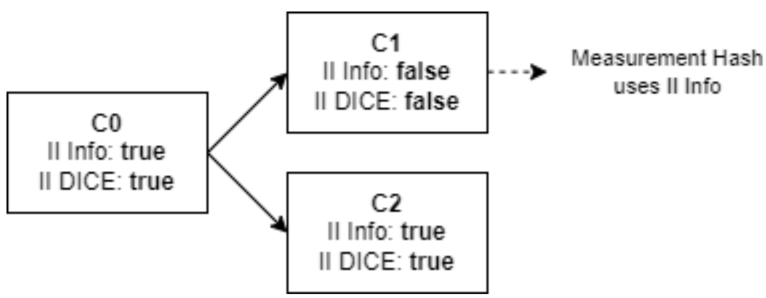


Figure 7: State of Context Tree after Inserting C2

6. From C0, derive a child context C3 with `uses_internal_input_info` set to `false`. The `uses_internal_input_info` flag in C0 will change to `false`.
7. Calculate a measurement hash using context C1, which now will not include the internal input info because the C0 flag is `false` again.

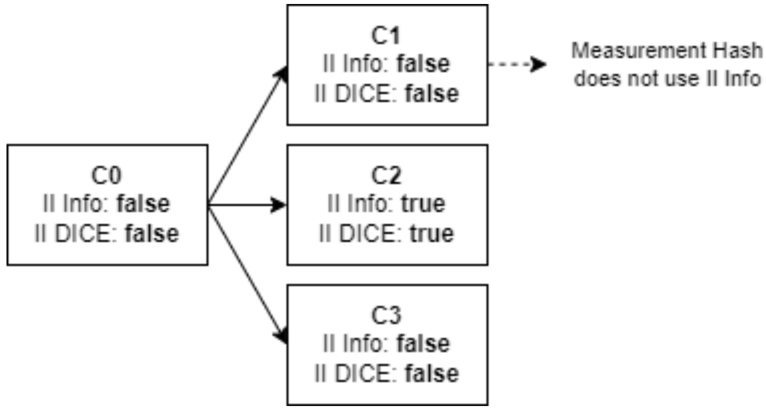


Figure 8: State of Context Tree after Inserting C3

The example attestation flow outlined in the DPE specification could be affected by these changes. The pseudocode below illustrates a modified version of the flow where an attacker is able to inject a `DeriveChild` command within the attestation flow sequence.

```
parent = dpe.InitializeContext(uds)
parent = dpe.DeriveChild(context, firmware0_hash)

context1, parent = dpe.DeriveChild(parent , firmware1_hash, uses-internal-input-info=false,
↪ retain-parent=true)
context1, cert_chain = dpe.CertifyKey(context1)

context2, parent = dpe.DeriveChild(parent , firmware2_hash, uses-internal-input-info=true,
↪ retain-parent=true) // Executed by separate thread/process
signature = dpe.Sign(context1, attestation_challenge)

attestation_response = cert_chain, signature
```

The resulting attestation response generated via the child context represented by `context1` would have a signature that cannot be validated by the returned public key.

Recommendation

Delete the two lines of code which set the parent context's `uses_internal_input_info` and `uses_internal_input_dice` flags within the implementation of the `DeriveChild` command.

Retest Results

2023-10-10 – Fixed

Pull request #202 fixed this finding per the above recommendation.

Medium

Timing Side-Channel Exposes Context Handles

Overall Risk	Medium	**Finding ID**	NCC-MSFT283-PTP
Impact	Medium	**Component**	DPE
Exploitability	Medium	**Category**	Data Exposure
		Status	Fixed

Impact

Knowledge of a context handle might enable an attacker to impersonate other entities within the same locality to the DPE command handler. While an attacker could also retrieve context handles from other localities, it was found these could not be used to execute DPE commands.

Description

The following function is called from `get_active_context_pos()`, which is invoked by most DPE commands. The purpose is to retrieve the handle's index in `DpeInstance::contexts[]`, where `ContextHandle`'s are stored.

```
fn get_active_context_pos_internal(
        &self,
        handle: &ContextHandle,
        locality: u32,
    ) -> Result<usize, DpeErrorCode> {
        let mut valid_handles = self
            .contexts
            .iter()
            .enumerate()
            .filter(|(_, context)| {
                context.state == ContextState::Active && &context.handle == handle
            })
            .peekable();
        if valid_handles.peek().is_none() {
            return Err(DpeErrorCode::InvalidHandle);
        }
        let mut valid_handles_and_localities = valid_handles
            .filter(|(_, context)| context.locality == locality)
            .peekable();
        if valid_handles_and_localities.peek().is_none() {
            return Err(DpeErrorCode::InvalidLocality);
        }
        let (i, _) = valid_handles_and_localities
            .find(|(_, context)| {
                context.state == ContextState::Active
                    && &context.handle == handle
                    && context.locality == locality
            })
            .ok_or(DpeErrorCode::InternalError)?;
```

Figure 9: caliptra-dpe/dpe/src/dpe_instance.rs:149-170

The code first creates an iterator over all `Active` context handles that match the `ContextHandle`. The comparison operation is automatically derived on the type, as can be seen below.

```
#[derive(Debug, PartialEq, Eq, Clone, Copy, zerocopy::AsBytes, zerocopy::FromBytes)]
pub struct ContextHandle(pub [u8; ContextHandle::SIZE]);
```

Figure 10: caliptra-dpe/dpe/src/context.rs:121-122

The automatically derived comparison does not execute in constant time[21]. By repeatedly observing the execution time with varying inputs, an attacker can determine the correct context handle in a byte-by-byte manner. As per the DPE specification draft[22], the context handle must be held secret (emphasis added by NCC Group):

> ***The context handle MUST be unguessable in practice.*** *If the context handle value is an index to a client's DPE context data, it SHOULD be random and at least 16 bytes in length. The reason for this is that a* ***context handle authorizes operations*** *on the associated context. So, for example, it's possible for parent and child components to share the same encrypted session, but the child should not be able to leverage that shared session to impersonate the parent.*

Note that only impersonation within the same locality is possible. While the context handle bytes for other localities could be leaked, the command handlers check the origin locality, and this cannot be spoofed.

Recommendation

Use constant_time_eq[23] or similar to compare context handles in constant time. Additionally, it would make sense to first filter context locality before comparing the handles.

Reproduction Steps

Observe timings for a command (e.g. `RotateCtx`) that looks up the index via above method. Recover the context handle byte by byte. In ideal scenario, the handle would be recovered in at most 256*16 (16 bytes to recover, each has 256 possible values) operations.

Depending on the implementation details, the actual comparisons could use larger types, which might make the attack more time consuming.

Retest Results

2023-10-11 – Fixed

Pull request #199 fixed this finding per the above recommendation.

21. Rust Lang slice cmp is not constant time.
22. TCG DPE Specification, Section 5.6 "Contexts"
23. Rust Lang const_time_eq

Medium

DestroyCtx Command Can Corrupt Context Tree

Overall Risk	Medium	**Finding ID**	NCC-MSFT283-KML
Impact	Medium	**Component**	DPE
Exploitability	High	**Category**	Other
		Status	Fixed

Impact

The `DestroyCtx` command could be used to corrupt the context tree structure, making it possible for context handles to have incorrect parents or children set, even from different localities. This flaw could be leveraged by an attacker to call the `CertifyKey` and `Sign` operations on a TCI that has the incorrect parent set, making it attest that it was derived from an entirely different context handle.

The impact is limited by the fact that the misused entries have to be created after `DestroyCtx` command is called.

Description

Contexts are stored in the `dpe.contexts[]` array with `MAX_HANDLES` (24) entries. As the contexts form a tree, their relations are stored in the `parent_idx` and `children` bitmasks.

Every node has a `parent_idx` set to the array index of the context that created it, except for the initial context, where it's set to `ROOT_INDEX`. The node's children are tracked with a `children` bitmask, so when a bit `n` is set, `dpe.contexts[n]` is one of the children.

This parent-child relationship tracking is not amended when a context is deleted, which can result in a vulnerability.

Shown below is the whole of the `DestroyCtx` command handler.

```
impl CommandExecution for DestroyCtxCmd {
    fn execute(
        &self,
        dpe: &mut DpeInstance,
        _env: &mut DpeEnv<impl DpeTypes>,
        locality: u32,
    ) -> Result<Response, DpeErrorCode> {
        let idx = dpe.get_active_context_pos(&self.handle, locality)?;
        let context = &dpe.contexts[idx];
        // Make sure the command is coming from the right locality.
        if context.locality != locality {
            return Err(DpeErrorCode::InvalidLocality);
        }

        let to_destroy = if self.flag_is_destroy_descendants() {
            (1 << idx) | dpe.get_descendants(context)?
        } else {
            1 << idx
        };

        for idx in flags_iter(to_destroy, MAX_HANDLES) {
            if idx >= dpe.contexts.len() {
```

```
                return Err(DpeErrorCode::InternalError);
            }
            dpe.contexts[idx].destroy();
        }
        Ok(Response::DestroyCtx(ResponseHdr::new(
            DpeErrorCode::NoError,
        )))
    }
}
```

Figure 11: caliptra-dpe/dpe/src/commands/destroy_context.rs:37-67

And the `destroy()` method of `Context`:

```
    /// Destroy this context so it can no longer be used until it is re-initialized. The
    ↪ default
    /// context cannot be re-initialized.
    pub fn destroy(&mut self) {
        self.tci = TciNodeData::new();
        self.has_tag = false.into();
        self.tag = 0;
        self.state = ContextState::Inactive;
        self.uses_internal_input_info = false.into();
        self.uses_internal_input_dice = false.into();
    }
```

Figure 12: caliptra-dpe/dpe/src/context.rs:99-108

The code is shown in full to demonstrate that these functions do not use `Context.parent_idx` or the `Context.children` bitmask. When a Context is destroyed, its parent stays intact, which means that the parent's `children` bitmask will still contain a bit indicating this child should be valid, when in fact it points to an `Inactive` context (which could later be populated). This could be abused to destroy context entries later created by any locality (see detailed example in Reproduction Steps).

The way the children are handled is also problematic. Unless a flag is set to destroy the children as well, they will also remain intact. Their `parent_idx` field will point to an `Inactive` context which could be used later. The `parent_idx` field is used by the `ChildToRootIter` iterator, which is then used in functions `get_tcb_nodes()` and `compute_measurement_hash()`. Those functions are called from command handlers of `CertifyKey` and `Sign` respectively.

Both vulnerabilities could be thought of as a type of use-after-free bug. The entry is freed (by `DestroyCtx`), but there is still a reference to it (in `children` mask or in `parent_idx`) and on a subsequent allocation, it can be reused through the old references.

Recommendation

Use a tree data structure that correctly keeps track of its node relations.

Reproduction Steps

There are at least three attack scenarios where data from another locality can be used or deleted.

Using `CertifyKey` With a Parent From Another Locality

1. Call `Init` to create a `ContextHandle::default` for the current locality
2. Call `DeriveChild { handle = default_handle, retain_parent = true }`; store returned handle as `child_handle`

3. Call `DeriveChild { handle = child_handle, retain_parent = true }`; store returned handle as `grandchild_handle`
4. Call `DestroyCtx { handle = child_handle }` – `parent_idx` of `grandchild_handle` now points to a destroyed handle
5. Wait for another locality to create a new entry in `dpe.children[]`
6. Call `CertifyKey { handle = grandchild_handle }`

`CertifyKey` will use a parent from another locality for their operation.

Using `Sign` With a Parent From Another Locality

1. *Same five steps as for `CertifyKey` above
2. Call `Sign { handle = grandchild_handle }`

`Sign` will use a parent from another locality for their operation.

Destroying Context Entries of Another Locality

The following sequence of commands will destroy context entries created by other entities:

1. Call `Init` to create a `ContextHandle::default` for the current locality
2. Call `DeriveChild { handle = default_handle, retain_parent = true }`; store returned handle
3. Many iterations of command `DeriveChild { handle = stored_handle, retain_parent = true }`
4. Call `DestroyCtx` commands with handles all those newly created children – `stored_handle` now has `children` mask populated, but those children were destroyed
5. Wait for other localities to create entries in `dpe.children[]`
6. Call `DestroyCtx { handle = stored_handle, flags = DESTROY_CHILDREN_FLAG_MASK }`

Retest Results

2023-10-12 – Fixed

Pull requests #200, #207 and #232 fixed this finding.

Medium

Multiple Ways to Exhaust DPE Context Handles

Overall Risk	Medium	**Finding ID**	NCC-MSFT283-29Q
Impact	Medium	**Component**	DPE
Exploitability	High	**Category**	Denial of Service
		Status	Fixed

Impact

An attacker that is able to send DPE commands to Caliptra could exhaust the space in the array that is used to keep track of the context handles. Since the array is shared across localities, any SoC agent could make Caliptra unusable for others.

Description

NCC Group identified three ways to fill `DpeInstance::contexts[]` in order to induce a denial of service.

1. Normal Usage of DeriveChild

The `DeriveChild` command can be repeatably used with the `retain_parent` option set. This will fill the array, but the entries could still be deleted from the same locality.

2. Abuse of DeriveChild

The `DeviceChild` command can create context handles for another locality, even a non-existent locality. This can make the entries undeletable unless the parents were retained, in which case `DestroyCtx` can be used on a parent with the `destroy_children` flag set.

3. Retired Entries

Retired context entries cannot be deleted. This is actually a known and documented bug, as shown below:

```
/// A child was derived from this context, but it was not retained. This will need to be
/// destroyed automatically if all of it's children have been destroyed. It is preserved
↳ for its
/// TCI data, but the handle is no longer valid. Because the handle is no longer valid, a
↳ client
/// cannot command it to be destroyed.
```

Figure 13: caliptra-dpe/dpe/src/context.rs:173-177

Recommendation

To limit the impact of these attacks across localities, Caliptra could count how many context entries a locality created and forbid further creation once a certain number is reached.

Retest Results

2023-10-13 – Fixed

This finding is fixed by pull requests #907, #944 and #962.

Low

Premature Context State Modification in DeriveChild

Overall Risk	Low	**Finding ID**	NCC-MSFT283-6Y9
Impact	Low	**Component**	DPE
Exploitability	Low	**Category**	Other
		Status	Fixed

Impact

The implementation of the `DeriveChild` command prematurely modified certain properties of the new or parent context before it was certain that the command would succeed. This could lead to the behaviour of the parent context changing when computing measurement hashes or invalidating parent contexts.

Because all of the state-changing statements occurred subsequent to validation of the supplied context handle and locality, this could not be abused to modify contexts to which the attacker did not already have accessed, and therefore judged to carry a low overall risk.

Description

The `DeriveChild` command was implemented within the `DeriveChildCmd::execute` function. Fragments of this function are reproduced below.

```
fn execute(&self, dpe: &mut DpeInstance, env: &mut DpeEnv<impl DpeTypes>, locality: u32,) ->
↳ Result<Response, DpeErrorCode> {
    // ... Snipped for brevity
    dpe.contexts[parent_idx].uses_internal_input_info = self.uses_internal_info_input().into();
    dpe.contexts[parent_idx].uses_internal_input_dice = self.uses_internal_dice_input().into();

    // ... Snipped for brevity

    // Make sure it can be the default if it is supposed to be.
    if self.makes_default() {
        let default_context_idx =
            dpe.get_active_context_pos(&ContextHandle::default(), target_locality);

        if !self.safe_to_make_default(parent_idx, default_context_idx) {
            return Err(DpeErrorCode::InvalidArgument);
        }
    }

    let child_handle = if self.makes_default() {
        ContextHandle::default()
    } else {
        dpe.generate_new_handle(env)?
    };

    if !self.retains_parent() {
        dpe.contexts[parent_idx].state = ContextState::Retired;
        dpe.contexts[parent_idx].handle = ContextHandle([0xff; ContextHandle::SIZE]);
    } else if !dpe.contexts[parent_idx].handle.is_default() {
        dpe.contexts[parent_idx].handle = dpe.generate_new_handle(env)?;
    }
```

```
    dpe.contexts[child_idx].activate(&ActiveContextArgs {
            context_type: ContextType::Normal,
    // ... Snipped for brevity

    dpe.add_tci_measurement(env, child_idx, &TciMeasurement(self.data), target_locality)?;

    // Add child to the parent's list of children.
    dpe.contexts[parent_idx].add_child(child_idx)?;
```

Figure 14: caliptra-dpe/dpe/src/commands/derive_child.rs:94-168

Within this function, the `self` reference pointed to the command received from the mailbox, and therefore contained potentially untrusted data. Once an appropriate parent context and child context had been determined, statements throughout the function modified individual properties of the parent or child context. These included:

- `dpe.contexts[parent_idx].uses_internal_input_info`
- `dpe.contexts[parent_idx].uses_internal_input_dice`
- `dpe.contexts[parent_idx].state`
- `dpe.contexts[parent_idx].handle`
- `dpe.contexts[child_idx].activate()`

Each of these statements were followed by one or more statements which could result in the abnormal termination of the function returning an error response, which would result in the contexts being placed into an inconsistent state. All state-changing and prematurely terminating operations have been highlighted in the above snippet.

Recommendation

Ensure that the all properties of the parent and child contexts are modified within a single block at the end of the `DeriveChildCmd::execute` function, after the completion of all operations which might potentially fail.

Retest Results

2023-10-11 – Fixed

Pull request #202 fixed this finding per the above recommendation.

Info

ChildToRootIter Infinite Loop

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-VF7
Impact	None	**Component**	DPE
Exploitability	None	**Category**	Denial of Service
		Status	Fixed

Impact

A denial of service is possible if `ChildToRootIter` is created with an incorrect `idx`. No instances of incorrect usage were found in the current source code, therefore this finding is provided for informational purposes only.

Description

A Rust `Iterator` trait is defined by its `next` function, which returns an `Option` bearing the next value in the sequence or `None` if no further values are available."

In the below code, the iterator returns an error in case `self.idx` is too large. However, since `self.done` is not set, on the following `.next()` invocation the same error will happen, resulting in an infinite loop.

```
impl<'a> Iterator for ChildToRootIter<'a> {
    type Item = Result<&'a Context, DpeErrorCode>;

    fn next(&mut self) -> Option<Result<&'a Context, DpeErrorCode>> {
        if self.done {
            return None;
        }
        if self.count >= MAX_HANDLES {
            self.done = true;
            return Some(Err(DpeErrorCode::MaxTcis));
        }
        if self.idx >= self.contexts.len() {
            return Some(Err(DpeErrorCode::InternalError));
        }
```

Figure 15: caliptra-dpe/dpe/src/context.rs:253-266

The callers of inspected code were found to not be able to reach the buggy behavior. Nevertheless, fixing this iterator would make it more robust against misuse.

Recommendation

The iterator implementation should set `self.done = true` like in the error case a few lines above.

Reproduction Steps

The following code can be used to trigger the infinite loop behavior.

```
        for _ in ChildToRootIter::new(30, &contexts) {
        }
```

Retest Results

2023-10-10 – Fixed

Fixed as per recommendation in pull request #198.

Info

Test Code May Panic on Some Inputs

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-97Q
Impact	None	**Component**	DPE
Exploitability	None	**Category**	Security Improvement Opportunity
		Status	Fixed

Impact

Test code could be used as a base for real implementation, and bugs could be propagated. It should be noted that towards the end of the assessment, code for `DpePlatform::get_certificate_chain` has been merged[24], and it does not contain the described bug.

Description

The `GetCertificateChain` command contains user controlled parameters `offset` and `size`. This used to be handled by a "not-implemented shim" in `DpePlatform::get_certificate_chain()`, but has been correctly implemented during the period of this assessment. However, there is still a `DefaultPlatform::get_certificate_chain()` function that contains a possible denial of service vulnerability, were it to be used. At the moment, this function is only used in code for tests.

The maximum values for `offset` and `size` are correctly checked. However, if the `size` value is less than `MAX_CHUNK_SIZE` or if the `offset` is set within the last `MAX_CHUNK_SIZE` bytes of the end of `TEST_CERT_CHAIN`, the source argument to `copy_from_slice` will be smaller than `out` is. This will make the code panic at the marked line below.

```
impl Platform for DefaultPlatform {
    fn get_certificate_chain(
        &mut self,
        offset: u32,
        size: u32,
        out: &mut [u8; MAX_CHUNK_SIZE],
    ) -> Result<u32, PlatformError> {
        let len = TEST_CERT_CHAIN.len() as u32;
        if offset >= len {
            return Err(PlatformError::CertificateChainError);
        }
        let cert_chunk_range_end = min(offset + size, len);
        out.copy_from_slice(&TEST_CERT_CHAIN[offset as usize..cert_chunk_range_end as usize]);
        Ok(cert_chunk_range_end - offset)
    }
```

Figure 16: caliptra-dpe/platform/src/default.rs:155-169

The reason is the behavior of `copy_from_slice`[25], which says *"This function will panic if the two slices have different lengths."*

Then, `panic!` in caliptra-runtime (as well as the ROM and FMC) is implemented in `handle_fatal_error`, which eventually ends up in an infinite loop.

24. Caliptra – Pull Request #717 - "Implement Platform::get_certificate_chain"
25. Rust Lang copy_from_slice documentation

Recommendation

Use code such as `DpePlatform::get_certificate_chain()` which correctly handles partial slice copying.

Retest Results

2023-10-10 – Fixed

The example code was fixed as part of pull request #169.

Info

Chunk Size and Certificate Size Misuse in GetCertificateChain Command

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-VKD
Impact	None	**Component**	DPE
Exploitability	None	**Category**	Security Improvement Opportunity
		Status	Fixed

Impact

This finding does not have an impact because the values of the misused constants are coincidentally the same. This finding is raised for informational purposes.

Description

In the below code, `self.size` is checked to be at most `MAX_CERT_SIZE`, but later it is used as a size of data to be copied into `cert_chunk`, which is `MAX_CHUNK_SIZE` bytes large (note the "CERT" vs. "CHUNK" in the constant name).

Because of Rust memory safety guarantees, this would not result in a buffer overflow, but rather, would cause a panic like the one described in NCC-MSFT283-97Q.

```
impl CommandExecution for GetCertificateChainCmd {
    fn execute(
        &self,
        _dpe: &mut DpeInstance,
        env: &mut DpeEnv<impl DpeTypes>,
        _locality: u32,
    ) -> Result<Response, DpeErrorCode> {
        // Make sure the operation is supported.
        if self.size > MAX_CERT_SIZE as u32 {
            return Err(DpeErrorCode::InvalidArgument);
        }

        let mut cert_chunk = [0u8; MAX_CHUNK_SIZE ];
        let len = env
            .platform
            .get_certificate_chain(self.offset, self.size, &mut cert_chunk)
```

Figure 17: caliptra-dpe/dpe/src/commands/get_certificate_chain.rs:18-33

In the inspected code, both `MAX_CERT_SIZE` and `MAX_CHUNK_SIZE` equal 2048, so this would not be an issue.

Recommendation

The first marked condition check above looks like it should be against `MAX_CHUNK_SIZE`.

Retest Results

2023-10-12 – Fixed

Note that the fix for a related issue NCC-MSFT283-97Q eliminates the potential for panic in the example code. Additionally, the pull request #231 changed the code to use the correct variable.

Info

Context Handles Not Rotated on Error

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-DBR
Impact	None	**Component**	DPE
Exploitability	Medium	**Category**	Policy Violation
		Status	Risk Accepted

Impact

It is assumed that context handle rotation is mandated by the DPE specification to reduce the risk of commands being replayed in encrypted sessions. Because encrypted sessions were not in use in the current Caliptra implementation, this risk would not be mitigated in any way by rotating context handles and the finding is reported for informational purposes only.

Although many types of failures might be expected to reoccur when the same command was replayed, certain failures caused by hardware glitches could potentially be replayed successfully at a future time, while other failures due to conditions that are true at one time but false at another time might also be replayed successfully. An attacker with the ability to replay such a message might use this to alter the state of the DPE to their advantage.

Description

The DICE Protection Environment specification stated the following with regard to the context handle:[26]

> The context handle MUST NOT remain valid after it has been used by a command. In other words, context handles are single use. New context handles are returned by a DPE within a response to a client so it can be used on a subsequent command. Once a context handle is provided to the DPE by a client, the context handle is invalidated by the DPE.

The specification did not explicitly state what should happen in the event of an error. However, in the event that an error did occur during processing of a DPE command, it was true that the context handle had already been "provided to the DPE".

The DPE source code initiated the generation of a new context handle during the processing of four commands: `CertifyKey`, `ExtendTci`, `Sign` and `TagTci`. The implementation of the `TagTci` command is shown below.

```
fn execute(
    &self,
    dpe: &mut DpeInstance,
    env: &mut DpeEnv<impl DpeTypes>,
    locality: u32,
) -> Result<Response, DpeErrorCode> {
    // Make sure this command is supported.
    if !dpe.support.tagging() {
        return Err(DpeErrorCode::InvalidCommand);
    }
    // Make sure the tag isn't used by any other contexts.
    if dpe.contexts.iter().any(|c| c.has_tag() && c.tag == self.tag) {
        return Err(DpeErrorCode::BadTag);
```

26. Trusted Computing Group: DICE Protection Environment Specification, version 1.0 revision 0.6

```
    }

    let idx = dpe.get_active_context_pos(&self.handle, locality)?;

    if dpe.contexts[idx].has_tag() {
        return Err(DpeErrorCode::BadTag);
    }

    // Because handles are one-time use, let's rotate the handle, if it isn't the default.
    dpe.roll_onetime_use_handle(env, idx)?;
    let context = &mut dpe.contexts[idx];
    context.has_tag = true.into();
    context.tag = self.tag;

    Ok(Response::TagTci(NewHandleResp {
        handle: context.handle,
        resp_hdr: ResponseHdr::new(DpeErrorCode::NoError),
    }))
}
```

Figure 18: caliptra-dpe/dpe/src/commands/tag_tci.rs:18-53

The context handle was rotated in the highlighted statement, after a set of validation statements which could result in early termination with an error code. As a result, any of the error conditions would not result in the rotation of the context handle. In this particular case, it is notable that the specifically highlighted error statement was returned in response to a condition which might be true at one time, but might subsequently be false after the deletion of a context bearing a specific conflicting tag.

Recommendation

Ensure that context handles are rotated both when a command completes successfully and when it terminates due to an error. An exception must be made when the error relates to an invalid context handle. The error response format must be adapted to permit the inclusion of the rotated context handle. Alternatively, if the risk associated with non-rotated context handles is judged to be sufficiently low, the wording of the DPE specification should be altered to clarify this case.

Retest Results

2023-10-13 – Fixed

This issue was tracked in #883, which was originally closed with the following reason:

> We decided to not do this since rotating a handle on an error would mean that the caller no longer has access to that handle, since if a DPE command fails, we do not return the handle. Since rotating the handle destructively mutates DPE's context array, the old handle is invalid, and the caller has no way to access the new context corresponding to the rotated handle.
>
> Another reason is that rotating the handle on error could be confusing in other ways such as if the input handle itself invalid or if the DPE command failed due to an error in generating the new handle.

Because no action was taken to correct the code, NCC Group recommended that the wording in the Caliptra documentation or DPE specification should be adjusted to clarify this edge case. Pull request #953 was created to document that this behavior is expected:

> This implementation guarantees that no internal DPE state is changed if a command fails for any reason. This includes Context Handle rotation; **single-use context handles are not rotated if a command fails**.
>
> On failure, DPE will only return a command header, with no additional command-specific response parameters. This is in line with the CBOR-based main DPE spec, which does not return a response payload on failure.

Info

CryptoBuf Can Be Partially Initialized

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-32H
Impact	Low	**Component**	DPE
Exploitability	None	**Category**	Data Exposure
		Status	Fixed

Impact

Partial initialization of the type makes it possible for memory contents to be exposed. This finding is reported for informational purposes, because no instance was found where this API was misused.

Description

As seen in the code snippet below, `CryptoBuf::new()` method first creates an empty `vec` (its type is `ArrayVec<u8, MAX_SIZE>`, which is inferred by the compiler), then copies data from `bytes[]` into it, and finally forcefully sets the size to `algs.size()`.

```
/// A common base struct that can be used for all digests, signatures, and keys.
pub struct CryptoBuf(ArrayVec<u8, { Self::MAX_SIZE }>);

impl CryptoBuf {
    pub const MAX_SIZE: usize = AlgLen::MAX_ALG_LEN_BYTES;

    pub fn new(bytes: &[u8], algs: AlgLen) -> Result<CryptoBuf, CryptoError> {
        let mut vec = ArrayVec::new();
        vec.try_extend_from_slice(bytes)
           .map_err(|_| CryptoError::Size)?;
        unsafe { vec.set_len(algs.size()) };
        Ok(CryptoBuf(vec))
    }
```

Figure 19: caliptra-dpe/crypto/src/signer.rs:39-51

There are a few cases where the sizes do not match:

- `bytes.len() > MAX_SIZE` – `vec.try_extend_from_slice(bytes)` fails and `CryptoError::Size` is returned
- `bytes.len() <= MAX_SIZE` – `vec.try_extend_from_slice(bytes)` copies the data to `vec`, and its length is set to `bytes.len()`
- `algs.size() == bytes.len()` – `vec.set_len(algs.size())` does nothing, the length is already set to this value
- `algs.size() < bytes.len()` – `vec.set_len(algs.size())` shortens the `vec`, same effect could be achieved with `vec.truncate()`
- `algs.size() > bytes.len()` – `vec.set_len(algs.size())` **forces the length to be increased; uninitialized data can now be accessed**
- Note: `algs.size()` is coded to be at maximum `AlgLen::MAX_ALG_LEN_BYTES` (which is equal to `MAX_SIZE`)

While NCC Group has not seen a usage of `CryptoBuf::new()` that would trigger the described vulnerability, this is a fragile API that could be misused.

Note that in addition to direct usage of `CryptoBuf`, there are also a few alias types:

- `HmacSig` in `caliptra-dpe/crypto/src/signer.rs:37`
- `OpensslPrivKey` in `caliptra-dpe/crypto/src/openssl.rs:75`
- `PrivKey` in `caliptra-dpe/crypto/src/openssl.rs:80`
- `Digest` in `caliptra-dpe/crypto/src/lib.rs:60`

Recommendation

Check `bytes.len()` matches `algs.size()` and error out if not. The `unsafe` call to `set_len()` can now be removed. If there are valid cases where the lengths do not match, they should be documented and handled safely.

Reproduction Steps

Behavior was tested with the following code snippet.

```
    let foo = [1u8; 1];
    println!("new: {:?}", CryptoBuf::new(&foo, AlgLen::Bit384).unwrap().bytes());
```

When the code is ran, we observed data additional to the provided value `1`. The data also changed on each run.

```
cryptobuf_test$ cargo run
new: [1, 48, 97, 51, 50, 57, 48, 48, 48, 32, 114, 119, 45, 112, 32, 48, 48, 48, 53, 51, 48,
↳48, 48, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
cryptobuf_test$ cargo run
new: [1, 0, 0, 0, 0, 0, 0, 0, 57, 48, 48, 48, 45, 53, 53, 54, 57, 100, 54, 54, 56, 97, 48, 48,
↳48, 32, 114, 119, 45, 112, 32, 48, 48, 48, 53, 51, 48, 48, 48, 32, 0, 0, 0, 0, 0, 0, 0, 0]
```

Retest Results

2023-10-11 – Fixed

Pull request #201 fixed this finding.

Info

DeriveChild Permits Context Handle to Coexist With the Default Context

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-DDK
Impact	Undetermined	**Component**	DPE
Exploitability	High	**Category**	Other
		Status	Fixed

Impact

The implementation does not conform to the specification. The impact of this was not assessed.

Description

Draft DPE specification[27] says (emphasis by NCC Group):

> A DPE SHOULD support default context(s) and may support only default context(s). If a DPE supports default contexts, it MUST support one default context per session. A DPE **MUST NOT allow simultaneous use of a default context and context handles** within the same session: these are mutually exclusive.

However, the implementation does not agree with this. When DPE already has a default context (this is the state after initialisation) and `DeriveChild` command is executed with `retain_parent = true`, `default = false`, none of the following statements will be executed. This makes is possible for a default context to remain, and for a new context handle to be generated.

```
        if !self.retains_parent() {
            dpe.contexts[parent_idx].state = ContextState::Retired;
            dpe.contexts[parent_idx].handle = ContextHandle([0xff; ContextHandle::SIZE]);
        } else if !dpe.contexts[parent_idx].handle.is_default() {
            dpe.contexts[parent_idx].handle = dpe.generate_new_handle(env)?;
        }
```

Figure 20: caliptra-dpe/dpe/src/commands/derive_child.rs:148-153

Recommendation

The `DeriveChild` command handler should not allow creation of a generated handle in case an existing default handle is retained.

Retest Results

2023-10-13 – Fixed

The pull requests #203 and #235 fixed this finding.

27. TCG DPE Specification – Section 5.6.1, "Default Contexts"

7 Finding Details – Drivers

Low

MailboxSendTxn drop() Handling Not Exhaustive

Overall Risk	Low	**Finding ID**	NCC-MSFT283-4CR
Impact	Low	**Component**	Drivers
Exploitability	Low	**Category**	Denial of Service
		Status	Fixed

Impact

The intention of `Drop for MailboxSendTxn` appears to be to return the mailbox into the `Idle` state, however, this will only happen if it was in `RdyForCmd` state at the start. This could potentially leave mailbox in a non-operational state following an error where the code handling it relied on the destructor to return the mailbox state machine to `Idle`. This finding represents a potential denial of service vulnerability.

Description

The `MailboxSendTxn` destructor is implemented in the code shown below.

```
impl Drop for MailboxSendTxn<'_> {
    fn drop(&mut self) {
        //
        // Release the lock by transitioning the mailbox state machine back
        // to Idle.
        //
        if self.state == MailboxOpState::RdyForCmd {
            //
            // Send dummy request to transition the state machine to execute state.
            //
            let _ = self.send_request(0, &[]);
            // Release the lock
            let _ = self.complete();
        }
    }
}
```

Figure 21: caliptra-sw/drivers/src/mailbox.rs:244-259

The full set of operational states is defined in the same file:

```
/// Malbox operational states
pub enum MailboxOpState {
    #[default]
    RdyForCmd,
    RdyForDlen,
    RdyForData,
    Execute,
    Idle,
}
```

Figure 22: caliptra-sw/drivers/src/mailbox.rs:24-32

It can be seen only `RdyForCmd` state is handled, however, transition to `RdyForDlen`, `RdyForData` and `Execute` states is possible from publicly accessible methods such as `write_cmd`, `write_dlen` and `execute_request` respectively, although there are indirect calls as well.

Recommendation

Handle transition from any state into `Idle`.

Retest Results

2023-10-10 – Fixed

Fixed with the pull request #856, which force unlocks the Mailbox in `drop()`.

Info

Random Number Generation Iterator Potentially Returning Non-Random Values

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-2QM
Impact	High	**Component**	Drivers
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

The Caliptra drivers provided an implementation of the `Iterator` trait which could be used to supply a sequence of a pre-defined quantity of random numbers, produced by the CSRNG peripheral connected to the device. Under certain conditions, this implementation could enter a state where the supplied values could not be guaranteed to be random.

This iterator was not used by the code in a way which could lead to this condition, and as a result this finding is being reported for informational purposes only. However, the condition was not documented or prohibited by the iterator code, leading to the possibility that future code might reuse the iterator code in a dangerous fashion, leading to the possibility that random values produced by the code might be more easily guessable by an attacker.

Description

A Rust `Iterator` trait is defined by its `next` function, which returns an `Option` bearing the next value in the sequence or `None` if no further values are available. The Caliptra drivers provided an implementation of this trait which could be used to produce a sequence of a pre-defined quantity of random numbers, produced by the CSRNG peripheral connected to the device.

```
fn next(&mut self) -> Option<Self::Item> {
    let csrng = self.csrng.regs();
    if self.num_words_left == 0 {
        None
    } else {
        if self.num_words_left % WORDS_PER_GENERATE_BLOCK == 0 {
            // Wait for CSRNG to generate next block of 4 words.
            wait::until(|| csrng.genbits_vld().read().genbits_vld());
        }

        self.num_words_left -= 1;

        Some(csrng.genbits().read())
    }
}
```

Figure 23: caliptra-sw/drivers/src/csrng.rs:209-223

The value of the constant `WORDS_PER_GENERATE_BLOCK` was equal to `4`, reflecting the fact that the underlying CSRNG hardware would produce random values in blocks of 4 words (or 16 bytes) at a time.

This function was developed on the assumption that the iterator would always be initialised with a `num_words_left` value which was a multiple of four. Only then would the condition `self.num_words_left % WORDS_PER_GENERATE_BLOCK == 0` be true on the first invocation of the `next` function. If the iterator had been initialised with any other value, then the initial result would not have taken into account the current state of the CSRNG peripheral, which might return uninitialised data.

Recommendation

Ensure that the `csrng.genbits_vld()` condition is awaited on the initial execution of the `next` method. This could be performed by implementing a forward-running counter which would either replace the existing `num_words_left` counter with a separate field to store the maximum number of words to be returned by this iterator.

If such a fix is not desired for any reason, then the iterator should check during initialisation whether the `num_words_left` value is divisible by four. If this is not the case, then the code should return an appropriate error code or panic, as appropriate.

Retest Results

2023-10-11 – Fixed

Pull request #470 fixed this finding. Additionally, changes in pull request #892 made sure the API cannot be misused.

Info

LMS Verifier Permitted Invalid q Value

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-NTF
Impact	None	**Component**	Drivers
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

The LMS signature validation function accepted an invalid value of the `q` parameter from a supplied signature. Because the specific edge case resulted in the operation failing at a later stage and returning a different error, there was no security impact and the finding is reported for informational purposes only.

Description

The LMS signature scheme[28] is based on a Merkle tree - a binary tree where each leaf node contains hash digest generated using a secure hash algorithm such as SHA-256. Each signature generated using the scheme contains a parameter `q`, which represents the index of the leaf node used to create the specific signature. Given a tree of a height `h`, leaf nodes are numbered within the range `0 ..= (2^h)-1`. An alternative numbering scheme exists that covers all nodes whether they are leaf or branch nodes. Known as the node number, the root node of the tree is assigned the node number of `1` while the nodes in successive levels of the tree with height `h` are labelled with a node number within the range of `2^h ..= 2^h + (2^h-1)`. The node number corresponding to any value `q` would thus be calculated as `2^h + q`, and the largest possible node number within a tree of a given height `h` would be `2^h + (2^h- 1)`, which could otherwise be expressed as `2^(h+1)-1`.

While verifying a submitted LMS signature using the `verify_lms_signature_cfi` function, the following code was used to validate the node number:

```
let (_, tree_height) = get_lms_parameters(lms_sig.tree_type)?;
let mut node_num: u32 = (1 << tree_height) + lms_sig.q.get();
if node_num > 2 << tree_height {
    return Err(CaliptraError::DRIVER_LMS_INVALID_PVALUE);
}
```

Figure 24: caliptra-sw/drivers/src/lms.rs:412-416

Having determined the tree height (which could be one of the values 5, 10, 15, 20 or 25), the node number `node_num` was calculated as `2^h + q` (using the equivalent left-shift operator to calculate the power of two). An error was then returned if this calculated value was greater than `2^(h+1)`. This implied that a calculated value which was *equal* to `2^(h+1)` being permitted, despite the fact that, as indicated earlier, the maximum possible node number should be `2^(h+1)-1`.

Presenting a concrete example given a tree height of 5, the node numbers should be within the range of `1 ..= 63`. A value of 64 should be rejected by the function, but would be permitted according to the above logic.

In practice, supplying a `q` value to trigger this edge case would result in the subsequent loop in lines 453-485 running an additional iteration and attempting to read the LMS signature's tree path out of bounds. Because this value was implemented as a Rust slice,

28. Internet Research Task Force: RFC 8554

this would result in an error and cause the signature verification function to fail with the `DRIVER_LMS_PATH_OUT_OF_BOUNDS` error rather than the `DRIVER_LMS_INVALID_PVALUE` which might be expected of an incorrect `q` value.

Recommendation

Correct the node number validation code to use the `>=` operator.

Retest Results

2023-10-11 – Fixed

Pull request #861 fixed this finding per the above recommendation.

Info

Comment and Code Mismatch in derive_cdi

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-GGG
Impact	None	**Component**	FMC
Exploitability	None	**Category**	Other
		Status	Fixed

Impact

Incorrect API description could lead to the API being used incorrectly. In the described case, there is potential for HMAC key and output CDI to be used in place of each other.

Description

The following function has incorrectly described parameters.

```
/// Permute Composite Device Identity (CDI) using Rt TCI and Image Manifest Digest
/// The RT Alias CDI will overwrite the FMC Alias CDI in the KeyVault Slot
///
/// # Arguments
///
/// * `env` - ROM Environment
/// * `hand_off` - HandOff
/// * `rt_cdi` - Key Slot that holds the current CDI
/// * `fmc_cdi` - Key Slot to store the generated CDI
fn derive_cdi(
    env: &mut FmcEnv,
    hand_off: &HandOff,
    fmc_cdi: KeyId,
    rt_cdi: KeyId,
) -> CaliptraResult<()> {
    // Compose FMC TCI (1. RT TCI, 2. Image Manifest Digest)
    let mut tci = [0u8; 2 * SHA384_HASH_SIZE];
    let rt_tci = Tci::rt_tci(env, hand_off);
    let rt_tci: [u8; 48] = okref(&rt_tci)?.into();
    tci[0..SHA384_HASH_SIZE].copy_from_slice(&rt_tci);

    let image_manifest_digest: Result<_, CaliptraError> =
       Tci::image_manifest_digest(env, hand_off);
    let image_manifest_digest: [u8; 48] = okref(&image_manifest_digest)?.into();
    tci[SHA384_HASH_SIZE..2 * SHA384_HASH_SIZE].copy_from_slice(&image_manifest_digest);

    // Permute CDI from FMC TCI
    Crypto::hmac384_kdf(env, fmc_cdi, b"rt_alias_cdi", Some(&tci), rt_cdi)?;
    report_boot_status(FmcBootStatus::RtAliasDeriveCdiComplete as u32);
    Ok(())
}
```

Figure 25: caliptra-sw/fmc/src/flow/rt_alias.rs:190-220

The signature of the `hmac384_kdf` function is shown below:

```
/// Calculate HMAC-384 KDF
///
/// # Arguments
///
/// * `env` - FMC Environment
```

```
    /// * `key` - HMAC384 key slot
    /// * `label` - Input label
    /// * `context` - Input context
    /// * `output` - Key slot to store the output
    pub fn hmac384_kdf(
        env: &mut FmcEnv,
        key: KeyId,
        label: &[u8],
        context: Option<&[u8]>,
        output: KeyId,
```

Figure 26: caliptra-sw/fmc/src/flow/crypto.rs:62-76

The code in `hmac384_kdf` suggests the comments are correct here, and are incorrect in `derive_cdi`.

The only caller of `derive_cdi` is shown below:

```
        // Derive CDI
        Self::derive_cdi(env, hand_off, input.cdi, KEY_ID_RT_CDI)?;
        report_boot_status(FmcBootStatus::RtAliasDeriveCdiComplete as u32);
```

Figure 27: caliptra-sw/fmc/src/flow/rt_alias.rs:59-61

Recommendation

Fix the comment to match the code. Argument order and comment order suggests this may have been a copy-paste error.

Since the arguments are just a `KeyId` (a key index), the output does not need to be mutable, hence it is missing a `mut` that would make it clear which of the two is the output. Consider annotating output/destination arguments to avoid such potential errors.

Retest Results

2023-10-10 – Fixed

The comments were corrected as a part of a bigger pull request #894.

Info

Insufficient Validation of Memory Addresses

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-QBA
Impact	None	**Component**	FMC
Exploitability	None	**Category**	Data Validation
		Status	Fixed

Impact

A memory address validation function checked whether a given address was within a supplied range, but incorrectly handled addresses near the end of the acceptable range.

The current codebase did not contain any invocations where this could be abused by an attacker, and therefore the current finding has been reported for informational purposes. However, future developments reusing this function could lead to potential vulnerabilities such as data exposure by reading from inappropriate addresses, or arbitrary code execution by writing to inappropriate addresses.

Description

The `validate_address` function allowed the caller to determine whether a provided memory address was within a given region of memory.

```
fn validate_address(&self, phys_addr: u32) -> bool {
    phys_addr >= self.start && phys_addr <= self.start + self.size
}
```

Figure 28: caliptra-sw/fmc/src/hand_off.rs:34-36

An address equal to the end of a region should not be considered to be a part of that region. However, this function checked whether the supplied address was less than *or equal* to the address of the end of the region, indicated by `self.start + self.size`. As a result, a single external address would be erroneously reported as being within the memory region.

Additionally, the `validate_address` function accepts only a `phys_addr` argument, but not a corresponding `phys_size`. Therefore, a structure that begins inside the valid range, but extends beyond the valid range would be mistakenly accepted as being valid.

This function was invoked from the `is_valid` function of two types representing distinct memory regions:

1. The `IccmAddress::is_valid` function was used to determine whether the entry point of the Runtime image was contained within the ICCM memory region (`caliptra-sw/fmc/src/hand_off.rs:196`). A similar check had already been performed by the ROM while loading the Runtime (`caliptra-sw/image/verify/src/verifier.rs:659-660`), using the half-open `Range` type which correctly excluded the address marking the end of the range. Therefore, it was not possible that this FMC memory range validation code could be reached using an invalid range.
2. The `DccmAddress::is_valid` function was used to determine whether the image manifest had been written to a valid address within the DCCM memory range (`caliptra-sw/fmc/src/hand_off.rs:298`). Because this checked only the start address of the manifest, it would accept not only a manifest beginning at the end of the DCCM range, but also an address near the end of the DCCM range that would result in a portion of the manifest overlapping the end of this range. However, it is acknowledged that the only code which set this manifest address was located in the ROM, which always initialised it to a fixed

address that ensured that the entirety of fixed-size manifest would be contained within the same range (`caliptra-sw/rom/dev/src/flow/cold_reset/fw_processor.rs:392`).

Recommendation

Ensure that the `validate_address()` function uses the less-than operator (`<`) when comparing the supplied address with the end of the range.

When validating an address intended for a set of bytes, ensure that the `validate_address()` function is adapted to accept the desired size as an input and validates the end of the range according to this size.

Retest Results

2023-10-10 – Fixed

Address validation code has been moved into `BoundedAddress`, and the semantic equivalent of the `validate_address()` from the original finding is `validate_addr()`.

```
pub fn validate_addr(addr: u32) -> Result<(), CaliptraError> {
    let addr = addr as usize;

    if addr % core::mem::align_of::<T>() != 0 {
        return Err(CaliptraError::ADDRESS_MISALIGNED);
    }
    let size = core::mem::size_of::<T>();
    if addr < B::ORG || size > B::SIZE || addr > B::ORG + (B::SIZE - size) {
        return Err(B::ERROR);
    }
    Ok(())
}
```

Figure 29: caliptra-sw/drivers/src/bounded_address.rs:53-64

The size of the type and also its alignment are also now validated.

High

TOCTOU in SHA-512 Accelerator Lock Acquisition

Overall Risk	High	**Finding ID**	NCC-MSFT283-YMG
Impact	High	**Component**	ROM
Exploitability	Medium	**Category**	Timing
		Status	Fixed

Impact

The SHA-512 Accelerator was a shared component which could be accessed by both the Caliptra firmware and the SoC, and could be used to compute SHA-384 (and SHA-512) digests of shared memory regions, in particular the SRAM used to back the Mailbox. Simultaneous use was restricted by means of a lock.

By exploiting a race condition within the ROM code used to acquire this lock, the SoC could claim ownership of the peripheral while the ROM believed it had exclusive access. The SoC could then cause digests of benign signed code to be computed in place of malicious unsigned code that had been supplied to the ROM, which would result in the ROM accepting the malicious code to be validly signed and proceed to execute it.

Description

The firmware verification process followed by the ROM computed eight SHA-384 digests used to validate different sections of the supplied Image Bundle, including the FMC and Runtime code. For each individual digest calculation, the code attempted to acquire a lock for the SHA-512 Accelerator peripheral by means of the following code:

```rust
pub fn try_start_operation(&mut self) -> Option<Sha384AccOp> {
    let sha_acc = self.sha512_acc.regs();

    if sha_acc.lock().read().lock() && sha_acc.status().read().soc_has_lock() {
        None
    } else {
        // We acquired the lock, or we already have the lock (such as at startup)
        Some(Sha384AccOp {
            sha512_acc: &mut self.sha512_acc,
        })
    }
}
```

Figure 30: caliptra-sw/drivers/src/sha384acc.rs:45-56

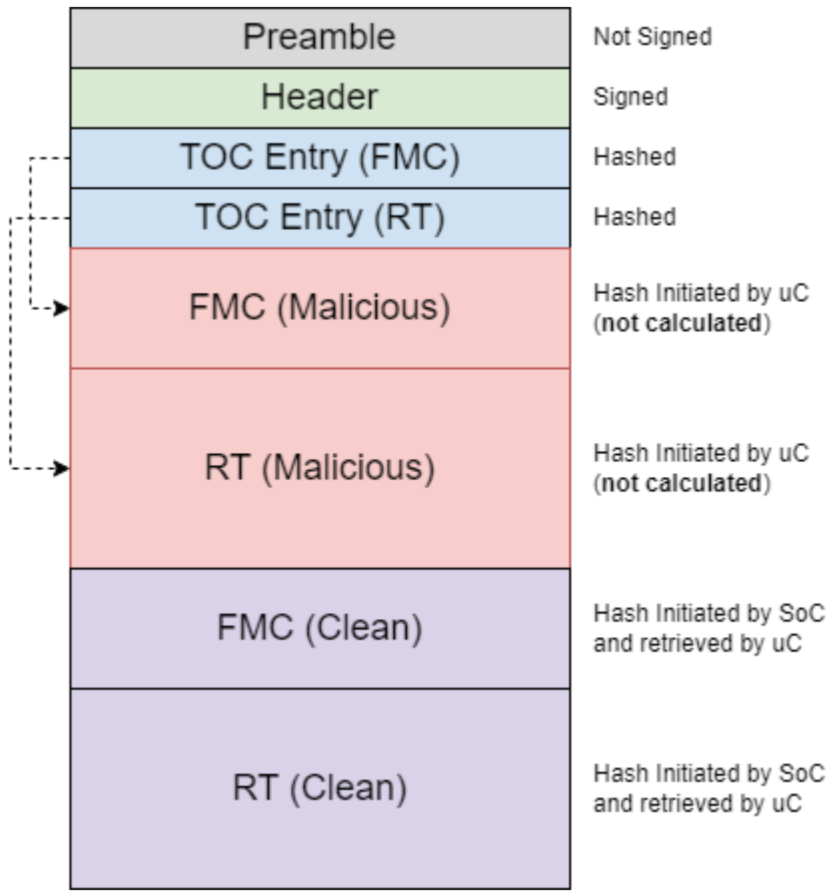
Two conditions could cause this operation to “fail”, which would result in the return of a `None` value and a subsequent attempt to acquire the same lock through an infinite loop wrapping this function. In order of testing, these conditions were:

1. Whether the lock had been acquired by some entity (through `sha_acc.lock().read().lock()` returning `true`).
2. Whether the acquired lock was owned by the SoC (through `sha_acc.status().read().soc_has_lock()` returning `true`). If this returned `false`, then the owner of the lock would be assumed to be the Caliptra microcontroller.

These conditions were determined by reading the LOCK and STATUS registers respectively of the SHA-512 Accelerator peripheral. If either of the conditions was not met, the firmware would consider that it owned the lock and proceed to request the calculation of a specific hash digest. However, the SoC could, through careful timing, ensure that the firmware believed that it owned the lock at the end of this check, while the lock was actually owned by the SoC. To do this, the SoC needed to:

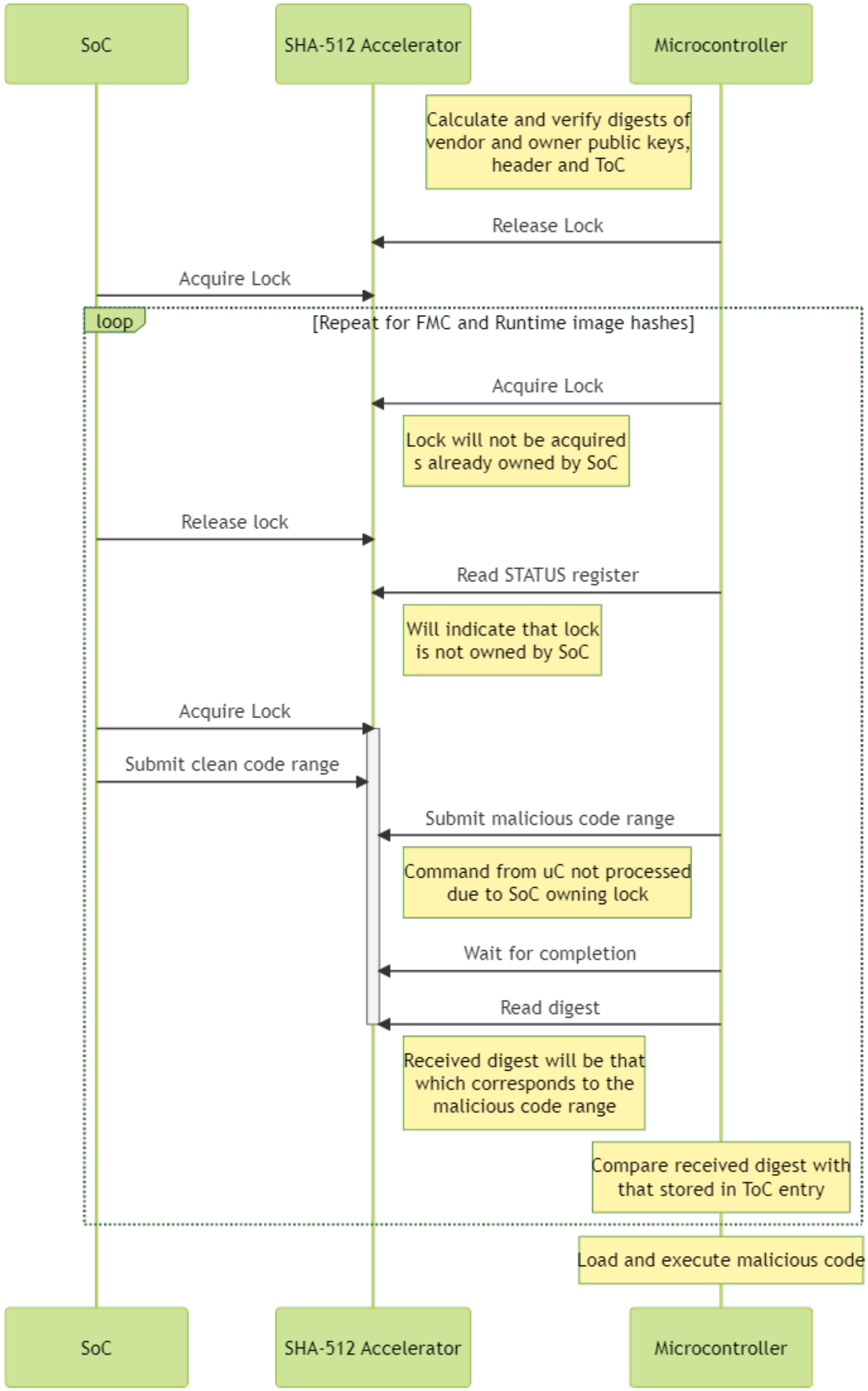
1. Acquire the lock prior to the first check
2. Release the lock prior to the second check
3. Re-acquire the lock after the second check

This could be used in an attack resulting in the execution of unsigned malicious code. The following example depends on a signed image bundle which has been tampered with by moving the clean FMC and Runtime images to a higher offset within the bundle and substituting malicious code in their place, as demonstrated in the diagram below.



The two ToC entries retained the offsets, sizes and hash digests of the original, clean, firmware images. However, when the ROM reached the point where the submitted firmware images themselves were to be validated, the SoC could, by releasing and re-acquiring the SHA-512 peripheral's lock as described earlier, cause the firmware to believe that it had submitted a request to calculate the digest of the image to be executed, while the resultant digest actually corresponded to a different memory region which contained the clean code but would not ultimately be executed.

The following sequence diagram illustrates the specific sequence that could to the malicious FMC and Runtime code being executed.


SoC
SHA-512 Accelerator
Microcontroller
Calculate and verify digests of vendor and owner public keys, header and ToC
Release Lock
Acquire Lock
loop
[Repeat for FMC and Runtime image hashes]
Acquire Lock
Lock will not be acquired s already owned by SoC
Release lock
Read STATUS register
Will indicate that lock is not owned by SoC
Acquire Lock
Submit clean code range
Submit malicious code range
Command from uC not processed due to SoC owning lock
Wait for completion
Read digest
Received digest will be that which corresponds to the malicious code range
Compare received digest with that stored in ToC entry
Load and execute malicious code
SoC
SHA-512 Accelerator
Microcontroller

An alternative way of exploiting this issue was also identified:

1. A SoC sends a malicious firmware through Mailbox
2. Lock is acquired as described above
3. SoC uses the streaming mode of SHA-512 Accelerator to calculate the digest of a clean firmware
4. Caliptra reads the `digest[]` register and the firmware verification succeeds

Recommendation

Because the check used in this function requires the reading of two independent registers, it may be impossible to prevent this condition from occurring within the supplied code. If possible, the hardware should be modified to ensure that the peripheral can be locked and its ownership checked in an atomic fashion.

Because the peripheral is forcibly locked at startup, the condition could be prevented in firmware by not releasing the lock prematurely (see finding "Premature Release of SHA-512 Accelerator Lock").

Retest Results

2023-10-09 – Fixed

With merged pull request #862 this finding is resolved. The code is changed so that the externally-accessible SHA-512 Accelerator is no longer used for image verification; instead a separate internal-only SHA-512 peripheral is used.

Additionally, `try_start_operation()` that is mentioned above was changed to include an `assumed_lock_state: ShaAccLockState` argument where the programmer specifies the current lock state. This would fix the issue if the provided state argument is correct. Currently, the only remaining usage of SHA-512 Accelerator is in FIPS self-test code.

Medium

Premature Release of SHA-512 Accelerator Lock

Overall Risk	Medium	**Finding ID**	NCC-MSFT283-3QD
Impact	Medium	**Component**	ROM
Exploitability	Low	**Category**	Denial of Service
		Status	Fixed

Impact

The SHA-512 Accelerator was a shared component which could be accessed by both the Caliptra firmware and the SoC, and could be used to compute SHA-384 (and SHA-512) digests of shared memory regions, in particular the SRAM used to back the Mailbox. Simultaneous use was restricted by means of a lock.

During the initial startup, Caliptra had exclusive access to this lock in order to perform its necessary work. By releasing this lock prematurely, this allowed another component to claim the lock and, by not releasing it, prevent the Caliptra ROM from completing its necessary validation of the First Mutable Code and Runtime and transferring control to those components.

The premature release of the lock could also facilitate the exploitation of a more severe TOCTOU condition described in NCC-MSFT283-YMG, which could result in the ROM loading unsigned FMC or Runtime code.

Description

The Caliptra ROM used the SHA-512 Accelerator to compute digests of various portions of the incoming Image Bundle. The first occurrence was observed in the code below, which was used to verify the public signing keys within the unsigned preamble of the image.

```
fn verify_vendor_pk_digest(&mut self) -> Result<(), NonZeroU32> {
    // ... Snipped for brevity
    let range = ImageManifest::vendor_pub_keys_range();

    let actual = self
        .env
        .sha384_digest(range.start, range.len() as u32)
        .map_err(|_| CaliptraError::IMAGE_VERIFIER_ERR_VENDOR_PUB_KEY_DIGEST_FAILURE)?;
```

Figure 31: caliptra-sw/image/verify/src/verifier.rs:279-298

A total of eight invocations of `ImageVerificationEnv::sha384_digest` were present within the same file, each executed sequentially within a single operation defined by the `ImageVerifier::verify` function.

The `sha384_digest` function itself was implemented as shown below:

```
impl<'a> ImageVerificationEnv for &mut RomImageVerificationEnv<'a> {
    /// Calculate Digest using SHA-384 Accelerator
    fn sha384_digest(&mut self, offset: u32, len: u32) -> CaliptraResult<ImageDigest> {
        loop {
            if let Some(mut txn) = self.sha384_acc.try_start_operation() {
                let mut digest = Array4x12::default();
                txn.digest(len, offset, false, &mut digest)?;
```

```
                return Ok(digest.0);
            }
        }
    }
```

Figure 32: caliptra-sw/rom/dev/src/verifier.rs:33-43

The highlighted `try_start_operation` function attempted to acquire a lock for the SHA-512 Accelerator and, if successful, returned a `Sha384AccOp` structure which implemented the `Drop` trait as shown below.

```
fn drop(&mut self) {
    let sha_acc = self.sha512_acc.regs_mut();
    sha_acc.lock().write(|w| w.lock(true));
}
```

Figure 33: caliptra-sw/drivers/src/sha384acc.rs:99-102

Writing a `true` value to this `lock` register had the effect of releasing the lock. From this point on, the SoC would be permitted to claim the lock for the SHA-512 Accelerator for itself and the ROM would be confined to the loop in the `sha384_digest` function shown above.

It is acknowledged that the watchdog timer was active during this period and would ultimately trigger a reset if such a condition occurred, transferring ownership of the lock to the ROM. To prevent the Caliptra firmware from starting, it would be necessary to continually re-acquire the lock to maintain the same condition.

Recommendation

Ensure that ownership of the SHA-512 Accelerator lock is maintained outside the context of any individual operation. As such, the lock should not be released during the `drop` function invoked when a specific instance of `Sha384AccOp` falls out of scope, but, at the earliest on the final completion of the `ImageVerifier::verify` function, when the peripheral is no longer required during the firmware startup process. It may be advisable to defer the release of the lock to the FMC or the beginning of the Runtime, in anticipation of any potential future modifications to those components that may require exclusive access to the peripheral.

Retest Results

2023-10-11 – Fixed

With merged pull request #862 this finding is resolved. The code is changed so that the externally-accessible SHA-512 Accelerator is no longer used for image verification; instead the separate internal-only SHA-512 peripheral is used.

Low

slice::fill(0) Does Not Always Zero Memory

Overall Risk	Low	**Finding ID**	NCC-MSFT283-962
Impact	Low	**Component**	ROM
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

Memory that is not zeroed stays resident for longer than strictly necessary, potentialy allowing it to be exposed by a separate memory exfiltration vulnerability.

Description

Zeroing memory with `slice::fill(0)` only guarantees the memory is read as zero when accessed again. If the compiler is not aware of any access after the zeroing, it might decide to remove the operation when optimising. This could leave the memory contents intact, and accessible through a wild pointer.

The important distinction here is that zeroing for cleanup is a different problem than zeroing for privacy[29]. Specifically, `slice::fill(0)` only zeroes for cleanup and since it does not use a volatile pointer in its implementation [30], it might be optimized-out if the compiler knows the value is not used anymore, which will often be the case for arrays on the stack.

There are multiple instances of `slice::fill(0)` being used to zero memory of a stack array and not used afterwards. The intention of that `fill(0)` is clearly to zero the memory for privacy. One such instance is listed below. While the data in any of the instances inspected was not deemed sensitive, the code should still execute these operations properly or alternatively just remove them.

```
fn derive_cdi(env: &mut RomEnv, measurements: &Array4x12, cdi: KeyId) -> CaliptraResult<()> {
        let mut measurements: [u8; 48] = measurements.into();
        Crypto::hmac384_kdf(env, cdi, b"fmc_alias_cdi", Some(&measurements), cdi)?;
        measurements.fill(0);
        report_boot_status(FmcAliasDeriveCdiComplete.into());
        Ok(())
    }
```

Figure 34: caliptra-sw/rom/dev/src/flow/cold_reset/fmc_alias.rs:99-106

This pattern also occurs in `MemoryRegions::zeroize()` (runtime/src/lib.rs:326-334) which is called from `FipsModule::zeroize()` (runtime/src/fips.rs:39). In this case there might be a specification requirement to zero the memory for privacy. Note that the `fill(0)` is behind the “zeroize” naming, which is commonly used to refer to “zeroing for privacy”.

Recommendation

Use the zeroize crate[31]. It is portable, embedded-friendly, and guarantees zeroing will not be optimized away.

Retest Results

2023-10-12 – Fixed

Pull requests #918 and #936 fixed this finding.

29. https://users.rust-lang.org/t/zeroing-a-slice-of-integers/33825/10
30. Rust Lang - slice and slice_spec
31. Rust Lang - Zeroize crate

Low

Buffer Overflow in PCR Logging

Overall Risk	Low	**Finding ID**	NCC-MSFT283-4DN
Impact	Undetermined	**Component**	ROM
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

An error in the PCR logging code can cause the log contents to overflow into the FUSE log. The impact of corrupting FUSE log was not determined.

Description

The relevant parts of `log_pcr` function are shown below.

```
pub fn log_pcr(
[...]
    if pcr_bank.log_index * core::mem::size_of::<PcrLogEntry>() > PCR_LOG_SIZE {
        return Err(CaliptraError::ROM_GLOBAL_PCR_LOG_EXHAUSTED);
    }
[...]
    let dst: &mut [PcrLogEntry] = unsafe {
        let ptr = PCR_LOG_ORG as *mut PcrLogEntry;
        let entry_ptr = ptr.add(pcr_bank.log_index);
        pcr_bank.log_index += 1;
        core::slice::from_raw_parts_mut(entry_ptr, 1)
    };

    // Store the log entry.
    dst[0] = pcr_log_entry;
```

Figure 35: caliptra-sw/rom/dev/src/pcr.rs:131,145-147,157-165

One can see the there is a check for `log_index`, but a value of `PCR_LOG_SIZE / core::mem::size_of::<PcrLogEntry>()` will pass the check, as the address of the start of a new log entry will be valid. However, the log entry will cross the `PCR_LOG_ORG+PCR_LOG_SIZE` and spill into FUSE_LOG.

For example, let's assume `log_index == 17` and `sizeof::<PcrLogEntry>() == 60`. The check passes because `17*60 == 1020` (not larger than `PCR_LOG_SIZE`, which is 1024). A `dst` slice is then constructed which spans from `PCR_LOG_ORG+1020` (0x500047FC) until `PCR_LOG_ORG+1020+sizeof::<PcrLogEntry>()` (0x50004838).

Memory layout shows the data immediately after PCR_LOG is FUSE_LOG.

```
pub const PCR_LOG_ORG: u32 = 0x50004400;
pub const FUSE_LOG_ORG: u32 = 0x50004800;
[...]
pub const PCR_LOG_SIZE: usize = 1024;
pub const FUSE_LOG_SIZE: usize = 996;
```

Figure 36: caliptra-sw/drivers/src/memory_layout.rs:34-35,62-63

Recommendation

The `log_index` check needs to be fixed to cover the newly created log entry as well.

Retest Results

2023-10-09 – Fixed

Fixed by using the newly introduced `PersistentDataAccessor` (pull requests #690, #704), which simplifies persistent data handling, and removes the reported and some other `unsafe` code instances.

Low

Critical Functions Not CFI Protected

Overall Risk	Low	**Finding ID**	NCC-MSFT283-BKC
Impact	Medium	**Component**	ROM
Exploitability	Low	**Category**	Configuration
		Status	Fixed

Impact

Gaps in Caliptra's CFI implementation may enable bypassing of anti-rollback protection.

Description

In the Caliptra ROM, Control Flow Integrity (CFI) is used to protect all functions which perform security-critical actions. The goal of CFI is to defend against attacks or exploits whose aim is to influence the firmware's call-graph.

Within the ROM, CFI is applied on an ad-hoc basis, by manually adding the `cfi_impl_fn` or `cfi_mod_fn` attributes to individual functions. At the moment, many sensitive operations are protected in this way, including the following:

- PCR extension and logging operations
- SHA1 hashing operations
- DICE operations including CDI derivations, UDS decryption, etc
- Manifest loading and signature verification

However, some sensitive functions are overlooked, such as the anti-rollback check which is shown below:

```
    fn svn_check_required(&mut self) -> bool {
        // If device is unprovisioned or if anti-rollback is disabled, don't check the SVN.
        !(self.env.dev_lifecycle() == Lifecycle::Unprovisioned || self.env.anti_rollback_disabl
        ↳ e())
    }
```

Figure 37: caliptra-sw/image/verify/src/verifier.rs:587-589

```
    fn anti_rollback_disable(&self) -> bool {
        self.soc_ifc.fuse_bank().anti_rollback_disable()
    }
```

Figure 38: caliptra-sw/image/verify/src/verifier.rs:101-103

Recommendation

Apply the `cfi_impl_fn` or `cfi_mod_fn` attributes to the above mentioned functions. Additionally, NCC Group wishes to point out that our sweep for CFI-worthy functions was incomplete due to time constraints. We encourage the Caliptra team to perform a deeper analysis for additional functions that warrant the protections offered by CFI.

Retest Results

2023-10-12 – Fixed

The code has been changed by pull request #911 to use the `cfi_assert!()` macro (`caliptra-sw/image/verify/src/verifier.rs:588-602`).

Info

Memory Not Cleared During Error Conditions

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-42W
Impact	None	**Component**	ROM
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

Several function cleanup operations intended to fill certain memory buffers with zeroes were bypassed if certain other functions returned error codes. Although it was observed that at least one such error could be triggered by an attacker controlling the mailbox, none of the specific affected values were determined to be sensitive. As a result, the finding has been reported as informational.

However, the risk could be upgraded in future developments which apply a similar pattern to more sensitive types of information.

Description

Several locations within the codebase failed to clear initialised memory blocks in the event of early termination due to error conditions. One example of this behaviour is shown below.

```
pub fn run(env: &mut RomEnv) -> CaliptraResult<Option<FirmwareHandoffTable>> {
    // ... Snipped for brevity
    let fmc_layer_input = dice_input_from_output(&ldevid_layer_output);

    // Download and validate firmware.
    let mut fw_proc_info = FirmwareProcessor::process(env)?;

    // Execute FMCALIAS layer
    FmcAliasLayer::derive(env, &fmc_layer_input, &fw_proc_info)?;
    ldevid_layer_output.zeroize();
    fw_proc_info.zeroize();
```

Figure 39: caliptra-sw/rom/dev/src/flow/cold_reset/mod.rs:53-75

The above code contained two lines which ended with `?` operators which, in the event that the preceding function returned an error, would result in the current function terminating immediately and propagating the error code. The subsequent cleanup calls to `ldevid_layer_output.zeroize()` and `fw_proc_info.zeroize()` would not run in the even of such an error, and subsequently the contents of those structures would remain in memory.

The `FirmwareProcessor::process` function referenced in the above code provided numerous other opportunities to fail, the first occurring in the following code, which invoked the `FirmwareProcessor::download_image` function:

```
let mut txn = Self::download_image(&mut env.soc_ifc, &mut env.mbox)?;
```

Figure 40: caliptra-sw/rom/dev/src/flow/cold_reset/fw_processor.rs:57

```
fn process_mailbox_commands<'a>(
    soc_ifc: &mut SocIfc,
    mbox: &'a mut Mailbox,
) -> CaliptraResult<ManuallyDrop<MailboxRecvTxn<'a>>> {
    soc_ifc.flow_status_set_ready_for_firmware();
```

```
    cprint!("[afmc] Waiting for Commands...");
    loop {
        if let Some(txn) = mbox.peek_recv() {
            match CommandId::from(txn.cmd()) {
                // .. Snipped for brevity
                _ => {
                    cprintln!("Invalid command 0x{:08x} received", txn.cmd());
                    txn.start_txn().complete(false)?;
                    return Err(CaliptraError::FW_PROC_MAILBOX_INVALID_COMMAND);
                }
```

Figure 41: caliptra-sw/rom/dev/src/flow/cold_reset/fw_processor.rs:126-172

This particular error would occur in response to an unexpected value in the mailbox command register. Because the mailbox was potentially externally controllable, this presented a means for an external entity to force a state in the ROM where data had not been cleared after execution.

Similar behaviour was present in several other code fragments, which are referenced in the Location section of this finding.

Recommendation

Ensure that sensitive data is cleared during both normal function execution flow and when the function is prematurely exited.

In the cases identified within this finding, this could be accomplished by implementing the `Drop` trait for the affected structures, which could call the `zeroize` method. Because ownership of these values are not passed to another entity, the `drop` method will always execute when the variable goes out of scope, which includes function exit due to an error.

If there is a need to keep the sensitive data within the buffer for only strictly as long as it is needed, then `drop` could be explicitly called or the data could be zeroed before the statements with the `?` operator, or by using more verbose language to ensure that the data is zeroed within each applicable execution branch which culminates in the end of function execution. Examples of this can already be found in other parts of the code, such as the following:

```
let result = Self::derive_cdi(env, &measurement, KEY_ID_ROM_FMC_CDI);
measurement.0.fill(0);
result?;
```

Figure 42: caliptra-sw/rom/dev/src/flow/cold_reset/fmc_alias.rs:59-61

Although the `zeroize` method is not used in this case, an equivalent method is highlighted which will achieve a similar effect, but should be considered in light of finding "slice::fill(0) Does Not Always Zero Memory".

Location

The following code locations point to a line that may prematurely exit a function via a `?` operator or another method. The subsequent line numbers in parentheses indicate the `zeroize` calls that are bypassed.

- `caliptra-sw/rom/dev/src/flow/cold_reset/fmc_alias.rs`: 82 (bypass 83)
- `caliptra-sw/rom/dev/src/flow/cold_reset/fmc_alias.rs`: 102 (bypass 103)
- `caliptra-sw/rom/dev/src/flow/cold_reset/fmc_alias.rs`: 185, 189, 192 (bypass 198, 212)

- `caliptra-sw/rom/dev/src/flow/cold_reset/idev_id.rs` : 248, 250, 269, 272, 275 (bypass 254, 285, 286)
- `caliptra-sw/rom/dev/src/flow/cold_reset/ldev_id.rs` : 186 (bypass 192, 207)
- `caliptra-sw/rom/dev/src/flow/cold_reset/mod.rs` : 70, 73 (bypass 74, 75)

Retest Results

2023-10-12 – Fixed

This finding is fixed by the pull requests #868, #877 and #936.

Info

ROM Integrity Test Does Not Cover .data Section

Overall Risk	Informational	**Finding ID**	NCC-MSFT283-BQM
Impact	None	**Component**	ROM
Exploitability	None	**Category**	Security Improvement Opportunity
		Status	Fixed

Impact

The ROM boot process includes an integrity test of the ROM itself that does not cover all of the sections. While the firmware does not currently use `.data`, if/when it does in future, this could become an oversight.

On integrity failure the boot process fails and on success the integrity hash is discarded (not used for any measurements).

Description

On ROM boot, as a part of the FIPS tests, the following integrity test is executed.

```
fn rom_integrity_test(env: &mut RomEnv, expected_digest: &[u32; 8]) -> CaliptraResult<()> {
    // WARNING: It is undefined behavior to dereference a zero (null) pointer in
    // rust code. This is only safe because the dereference is being done by an
    // an assembly routine ([`ureg::opt_riscv::copy_16_words`]) rather
    // than dereferencing directly in Rust.
    #[allow(clippy::zero_ptr)]
    let rom_start = 0 as *const [u32; 16];

    let n_blocks = unsafe { &CALIPTRA_ROM_INFO as *const RomInfo as usize / 64 };
    let digest = unsafe { env.sha256.digest_blocks_raw(rom_start, n_blocks)? };
    cprintln!("ROM Digest: {}", HexBytes(&<[u8; 32]>::from(digest)));
    if digest.0 != *expected_digest {
        cprintln!("ROM integrity test failed");
        return Err(CaliptraError::ROM_INTEGRITY_FAILURE);
    }
    Ok(())
}
```

Figure 43: caliptra-sw/rom/dev/src/main.rs:158-174

It calculates the SHA256 digest over the firmware, from `0x0` (ROM is loaded at this address) to `CALIPTRA_ROM_INFO`. Similar code can also be seen in `elf2rom()` (`caliptra-sw/builder/src/lib.rs`), where the ROM image is generated.

In the linker script, we can see there is actually more data stored in the ROM after `CALIPTRA_ROM_INFO`.

```
        . = ALIGN(64);
        CALIPTRA_ROM_INFO = .;
    } > ROM

    .data : ALIGN(4)
    {
        _sidata = LOADADDR(.data);
```

```
      _sdata = .;

      /* Must be called __global_pointer$ for linker relaxations to work. */
      PROVIDE(__global_pointer$ = . + 0x800);

    *(.sdata .sdata.* .sdata2 .sdata2.*);
      *(.data .data.*);

    . = ALIGN(4);
      _edata = .;
  } > DATA AT> ROM
```

Figure 44: caliptra-sw/rom/dev/src/rom.ld:75-92

This means that integrity of the initial `.data` section will not be checked by `rom_integrity_test()`.

While it is unusual for Rust code to have static or global initialised data (which would go into `.data` section), 'objdump' confirmed the section was not empty when compiled for x86-64. However, when compiled for riscv32imc, `.data` was empty. There are even indications that a `.data` section would not work correctly with the current code because copying it from ROM to DCCM is commented out[32].

Recommendation

Modify the linker script to move `CALIPTRA_ROM_INFO` after all the other sections that are present in ROM.

Retest Results

2023-10-09 – Fixed

Fixed with the pull request #928, as per the above recommendation.

32. Caliptra startup code does not copy `.data` section

Low

TOCTOU Condition in File Read Leading to Uninitialised Memory Buffer

Overall Risk	Low	**Finding ID**	NCC-MSFT283-VH3
Impact	Medium	**Component**	libcaliptra
Exploitability	Low	**Category**	Timing
		Status	Fixed

Impact

Two instances of C code were observed which read the contents of a file without sufficiently checking whether the number of bytes read matched the size of the allocated buffer. This could result in the buffer partially containing uninitialised data, which in turn could potentially leak sensitive information from previously freed memory within the process address space.

One instance of the code was within a file used for automated testing purposes. However, the other instance existed within an example project intended to instruct other developers in integrating the Caliptra API with their products. Within that project, the function was used to load a copy of the ROM and a copy of the FMC. Although it is expected that the integrity of both would subsequently be verified by Caliptra and thus rejected if found to contain uninitialised data, other concerns could emerge if developers were to re-use this example code in other contexts posing greater risk.

Description

The “hwmodel” example code contained a function intended to read the contents of a file into a memory buffer. This code is reproduced below:

```
static struct caliptra_buffer read_file_or_exit(const char* path)
{
    // Open File in Read Only Mode
    FILE *fp = fopen(path, "r");
    if (!fp) {
        printf("Cannot find file %s \n", path);
        exit(-ENOENT);
    }

    struct caliptra_buffer buffer = {0};

    // Get File Size
    fseek(fp, 0L, SEEK_END);
    buffer.len = ftell(fp);
    fseek(fp, 0L, SEEK_SET);

    // Allocate Buffer Memory
    buffer.data = malloc(buffer.len);
    if (!buffer.data) {
        printf("Cannot allocate memory for buffer->data \n");
        exit(-ENOMEM);
    }

    // Read Data in Buffer
```

```
    fread((char *)buffer.data, buffer.len, 1, fp);

    return buffer;
}
```

Figure 45: caliptra-sw/libcaliptra/examples/hwmodel/interface.c:27-54

A function named `read_file_or_die` with identical contents was also present at `caliptra-sw/hw-model/c-binding/examples/smoke_test.c:10-37`.

This code performed the following operations:

- Using `fopen`, open a file in read-only mode
- Using `fseek` and `ftell`, determine the size of the file
- Using `malloc`, allocate a buffer in memory using the length
- Using `fread`, read the file data into the buffer

Although the `fread` function accepted the full size of the allocated buffer, its return value, which would have indicated the number of bytes actually read, was never checked. In the event that the full expected data was no longer available, this meant that the `fread` function would not have modified the buffer beyond what was available. This subsequent portion of the buffer would therefore contain uninitialised data.

Recommendation

Check the return value of the `fread` function. If this value does not equal the size of the allocated memory buffer, then free the buffer and return an error.

Retest Results

2023-10-09 – Fixed

Pull request #896 fixed this finding per the above recommendation.

11 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.

Rating	Description
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.

Rating	Description
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.

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

Rating	Description
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.

Category Name	Description
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.

12 Provided Materials

To facilitate this engagement, NCC Group leveraged the following public source code and documentation resources.

Source Code

Caliptra ROM, FMC, Firmware

- https://github.com/chipsalliance/caliptra-sw (release tag release_v20230831_0)
- https://github.com/chipsalliance/caliptra-dpe (commit 76528b046e)

Caliptra RTL

https://github.com/chipsalliance/caliptra-rtl (commit 76d7c90fc8)

Documentation

Main Specification

https://github.com/chipsalliance/Caliptra/blob/f3ba3eaff457b66d53160a5b96136f32607304c3/doc/Caliptra.md

ROM Specification

https://github.com/chipsalliance/caliptra-sw/blob/1bf2a1b600296da11c9c7ce7fb9115c4225e385e/rom/dev/README.md

FMC Specification

https://github.com/chipsalliance/caliptra-sw/tree/1bf2a1b600296da11c9c7ce7fb9115c4225e385e/fmc#readme

Runtime Firmware Specification

https://github.com/chipsalliance/caliptra-sw/blob/1bf2a1b600296da11c9c7ce7fb9115c4225e385e/runtime/README.md

Hardware Specification

https://github.com/chipsalliance/caliptra-rtl/blob/76d7c90fc8eab682519676e12d3e1599040df43b/docs/Caliptra_Hardware_Specification.pdf

Integration Specification

https://github.com/chipsalliance/caliptra-rtl/blob/76d7c90fc8eab682519676e12d3e1599040df43b/docs/Caliptra_Integration_Specification.pdf

DICE Attestation Architecture Specification

https://trustedcomputinggroup.org/wp-content/uploads/TCG_DICE_Attestation_Architecture_r22_02dec2020.pdf

DICE Protection Environment Specification

https://trustedcomputinggroup.org/wp-content/uploads/TCG-DICE-Protection-Environment-Specification_14february2023-1.pdf