powHSM Security Assessment

IOV Labs

Version 1.1 – October 3, 2022

Prepared By
Paul Bottinelli
Kevin Henry

Prepared For
Bernardo Codesido
Adrian Eidelman
Ariel Mendelzon

1 Executive Summary

Synopsis

In June 2022, IOV Labs engaged NCC Group to perform a review of powHSM. Per the project documentation: "*Its main role is to safekeep and prevent the unauthorized usage of each of the powPeg's members' private keys. powHSM is implemented as a pair of applications for the Ledger Nano S, namely a UI and a Signer, and it strongly depends on the device's security features to implement the aforementioned safekeeping.*". In total, two consultants contributed 20 person days of effort over approximately five weeks. The assessment primarily focused on source code review, supplemented by 2 Ledger Nano S devices provided by IOV to facilitate testing.

In September 2022, the same consultants reviewed an updated version of the library addressing the findings in this report. In general, all findings and major comments were addressed by IOV and all documented findings are considered fixed.

Scope

The scope of the review includes github.com/rsksmart/rsk-powhsm/tree/3.0.0, targeting the `3.0.0` tagged release. The `UI` and `signer` components were identified as the highest priority, including common code in *src/common*. In particular, the following goals were identified by IOV to guide the review:

1. Seed cannot be extracted from the device
2. Signature operation authorization cannot be bypassed
   a. Signer update authorization
   b. Transaction signing
3. Recovery mode cannot be accessed without wiping the device first
4. An arbitrary app cannot be successfully used without wiping the device first
5. Arbitrary BIP32 paths cannot be used (either for signing or extracting the public key)
6. Blockchain state cannot be manipulated without the corresponding PoW

The subsequent re-test focused on changes made in the `3.0.1` tagged release.

Limitations

While physical devices were provided as part of this assessment, a review of the physical security of the Ledger Nano S and the included secure element was not in scope. This includes the investigation of side channel attacks, or attempts to reverse engineer the behavior of the secure element.

Key Findings

The assessment uncovered a number of low severity findings, including:

- Inconsistent Threshold Signature Validation Criteria: The order in which signers are authorized affects the result of the authorization process.
- Potentially Unsafe Exception Handling: Failure to use TRY-CATCH blocks appropriately may lead to incorrect or unpredictable behavior based on compiler optimizations.
- Failure to Validate Signer Authorizer Array Size May Lead to Out-of-Bound Memory Access: If a large number of distinct authorizers are provided, then the signer authorization process may write to out-of-bound memory addresses.

Additional Content

In addition to the formal findings listed above, the document contains sections detailing general comments and engagement notes, as well as a detailed walkthrough of review goals, with several more in-depth observations.

2 Dashboard

Target Data

Name	powHSM
Type	Embedded Application
Platforms	Embedded C; Ledger Nano S
Environment	Local Instance

Engagement Data

Type	Security Assessment
Method	Code-assisted
Dates	2022-06-06 to 2022-07-08
Consultants	2
Level of Effort	20 days

Targets

rsk-powhsm	https://github.com/rsksmart/rsk-powhsm/tree/3.0.0

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	0	
Low issues	5	🟨🟨🟨🟨🟨
Informational issues	1	⬜
Total issues	**6**	

Category Breakdown

Authentication	1	🟨
Cryptography	1	🟨
Data Validation	2	🟨🟨
Error Reporting	1	🟨
Other	1	⬜

Component Breakdown

UI	2	🟨🟨
UI, tcpsigner	1	🟨
signer	2	🟨⬜
signer, signer-certificate	1	🟨

🟪 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Inconsistent Threshold Signature Validation Criteria	Fixed	6PU	Low
Potentially Unsafe Exception Handling	Fixed	AP2	Low
Block Number Validation in Blockchain State Update Does Not Match Documentation	Fixed	2GR	Low
Failure to Validate Signer Authorizer Array Size May Lead to Out-of-Bound Memory Access	Fixed	W4M	Low
Onboarding State May Not Be Correctly Tracked	Fixed	7DB	Low
Flash Memory Endurance Considerations	Fixed	4GK	Info

4 Finding Details

Low Inconsistent Threshold Signature Validation Criteria

Overall Risk	Low	**Finding ID**	NCC-E003945-6PU
Impact	Low	**Component**	UI
Exploitability	Medium	**Category**	Cryptography
		Status	Fixed

Impact

The order in which signers are authorized affects the result of the authorization process. The presence of an invalid signer within the first *m* signers of the "*m-of-n*" threshold scheme will cause a failure, but the presence of an invalid signer after the first *m* signers will not. This property is not typical of most threshold signature schemes.

Description

The function `do_authorize_signer(volatile unsigned int rx, sigaut_t* sigaut_ctx)` implements the signer authorization protocol using an "*m-of-n*" threshold approach. This function is called multiple times on various messages using the same authorization context `sigaut_ctx` to track the number of valid authorization signatures. The first step involves initializing the `sigaut_ctx` with the correct signer information. The authorization context is defined in *signer_authorization.h*:

```
// Signer authorization context
typedef struct {
    sigaut_stage_t stage;

    sigaut_signer_t signer;

    bool authorized_signer_verified[MAX_AUTHORIZERS];

    union {
        cx_sha3_t auth_hash_ctx;
        cx_ecfp_public_key_t pubkey;
    };

    union {
        uint8_t buf[AUX_BUFFER_SIZE];
        uint8_t auth_hash[HASHSIZE];
    };
} sigaut_t;
```

Once initialized, `do_authorize_signer()` will iterate over the known authorizers looking for a key that validates the expected signature; see *signer_authorization.c*:

```
        // Check to see whether we find a matching authorized signer
        signature_valid = 0;
        for (int i = 0; i < TOTAL_AUTHORIZERS && !signature_valid; i++) {
           // Clear public key memory region first just in case initialization
           // fails
           explicit_bzero(&sigaut_ctx->pubkey, sizeof(sigaut_ctx->pubkey));
           // Init public key
           cx_ecfp_init_public_key(CX_CURVE_256K1,
258                                    authorizers_pubkeys[i],
259                                    sizeof(authorizers_pubkeys[i]),
260                                    &sigaut_ctx->pubkey);
261             signature_valid = cx_ecdsa_verify(&sigaut_ctx->pubkey,
262                                       0,
263                                       CX_NONE,
264                                       sigaut_ctx->auth_hash,
265                                       HASHSIZE,
266                                       APDU_DATA_PTR,
267                                       APDU_DATA_SIZE(rx));
268             // Cleanup
269             explicit_bzero(&sigaut_ctx->pubkey, sizeof(sigaut_ctx->pubkey));
270 
271             // Found a valid signature?
272             if (signature_valid) {
273                 sigaut_ctx->authorized_signer_verified[i] = true;
274             }
275         }
```

If the a valid signature is found, the result is stored within the current context. If no valid signature was found, then an error is thrown and the context is reset (zeroed out).

```
283         if (!signature_valid) {
284            // Invalid signature given, start over
285            reset_signer_authorization(sigaut_ctx);
286            THROW(SIG_AUT_INVALID_SIGNATURE);
287         }
```

Otherwise, the function proceeds to count the number of verified signatures in the current context, and will return either `SIG_AUT_OP_SIGN_RES_SUCCESS` if the threshold is met, or `SIG_AUT_OP_SIGN_RES_MORE`, indicating additional signatures are needed.

Note that the above behavior is not typical of threshold schemes, particularly because it may return an error even when the threshold is met. Consider a situation in which three signatures are provided, and 2-of-3 signatures are required to reach the threshold. Now consider a situation in which a single signature is corrupted or malicious. If the first or second signature is corrupted, the check on line 283 will cause an error to be thrown, the context to be wiped, and the signer authorization protocol will fail. However, if the third signer is corrupted, then the function will correctly validate the first signature, followed by the second, and then return `SIG_AUT_OP_SIGN_RES_SUCCESS` as the threshold has been met.

Recommendation

The result of the signer authorization protocol should not depend on the order in which signatures are validated. The protocol could be updated to proceed on a failed signature validation, and return success when the threshold is met. Alternatively, if a single invalid signature is intended to cause an error, then the complete set of signatures must be tested, rather than returning an early success. In either case, the behavior should be clearly documented.

Retest Results

2022-09-08 – Fixed

As part of release 3.0.1, commit 4967855 removed the conditional block of code that aborted the authorization process if any single signature was invalid. Therefore, the validation criteria should no longer depend on the order in which signatures are validated. This change is in line with NCC Group's recommendation, and the issue is therefore considered fixed.

Low

Potentially Unsafe Exception Handling

Overall Risk	Low	**Finding ID**	NCC-E003945-AP2
Impact	Undetermined	**Component**	signer, signer-certificate
Exploitability	Undetermined	**Category**	Error Reporting
		Status	Fixed

Impact

Failure to use TRY-CATCH blocks appropriately may lead to incorrect or unpredictable behavior based on compiler optimizations.

Description

Current guidance for Ledger app developers suggests that exceptions should be avoided for cryptographic code:

> Rationale: the exception mechanism is not standard C, and is difficult to use, particularly for error management. There were errors related to this exception model in every single app we reviewed in 2020.

Despite the above guidance, exception handling is required prior to SDK version 2.0. Proper usage of the SDK's exception handling model is documented with the following guidance:[1]

1. You must be careful to always close a `TRY` context when jumping out of it. ...the `CLOSE_TRY` macro must be used to close the `TRY` context before returning from the function...
2. When modifying variables within a `TRY` / `CATCH` / `FINALLY` context, always declare those variables `volatile`. This will avoid the compiler making invalid assumptions when optimizing your code because it doesn't understand how our exception model works.

The file *ledger/src/signer-certificate/src/main.c* contains a return within a `TRY` block without calling `CLOSE_TRY` first:

```
585     case RSK_END_CMD: // return to dashboard
586         os_sched_exit(0);
587         return;
588         // goto return_to_dashboard;
```

As per the recommendation above, the `CLOSE_TRY` macro should be called immediately prior to the return on line 587.

Several instances of local variables being modified within a `TRY` block without being declared `volatile` were also observed:

- In *ledger/src/signer/src/hsm-ledger.c* on line 30, the variable `tx` is declared as `unsigned int tx = 0;`, and later modified within a `TRY` block on lines 44 and 47.
- In *ledger/src/signer/src/sign.c* on line 54 the variable `pubkey_size` is declared as `int pubkey_size;`, and later modified within a `TRY` block on line 70.
    - In the same file on line 121 the variable `sig_size` is declared as `int sig_size;` and modified in a `TRY` block on line 137.

1. https://developers.ledger.com/docs/nano-app/troubleshooting/#exception-handling

In *ledger/src/signer-certificate/src/main.c* on line 457, the variable `index` is declared as `unsigned int index;`, and later modified within a `TRY` block on line 491.

As per the second recommendation above, each of these variables should be marked as `volatile`.

Recommendation

1. Add the missing `CLOSE_TRY` macro.
2. Add `volatile` to the listed variable declarations.

Location

- *ledger/src/signer-certificate/src/main.c*, lines 585-588.
- *ledger/src/signer/src/hsm-ledger.c*, line 30.
- *ledger/src/signer/src/sign.c*, lines 54, 121.
- *ledger/src/signer-certificate/src/main.c*, line 457.

Retest Results

2022-09-08 – Fixed

In the 3.0.1 release:

- The `signer-certificate` component has been removed, negating any instances of unsafe code in this component.
- In *hsm-ledger.c*, in the function `hsm_ledger_main_loop()`, the variable `tx` is now declared volatile.
- In *sign.c*, in the function `do_pubkey()`, the variable `pubkey_size` is now declared volatile, as are other local variables `private_key_data`, `private_key`, and `public_key`.
- Also in *sign.c*, in the function `do_sign()`, variables `private_key_data`, `private_key`, and `sig_size` are now declared volatile.

These combined changes either remove or address the identified unsafe instances of exception handling, thereby addressing this finding.

Low

Block Number Validation in Blockchain State Update Does Not Match Documentation

Overall Risk	Low	**Finding ID**	NCC-E003945-2GR
Impact	Medium	**Component**	signer
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

Failure to ensure the block numbers are sequential may result in an attacker being able to induce the HSM in a bad state.

Description

The powHSM implementation provides users with the ability to perform a number of operations, including the security-critical `advanceBlockchain` operation, which advances the HSM state to a newer state. For the process of advancing the blockchain state, an array of `m` blocks is provided as input to the process. These blocks correspond to the consecutive (confirmed) heads of the blockchain, and as such they include various fields, including their parent hash, some Proof-of-Work values and the block number, represented as a 32-bit unsigned integer.

The algorithm defined in the file *blockchain-bookkeeping.md* defines a number of checks and validation rules to ensure the state can only be advanced to a legitimate blockchain block. Among these checks, the algorithm states the following regarding the ordering of the blocks by their block number:

> This blocks array is indexed from 0 to m-1 and blocks must be ordered from newest to oldest, i.e. `blocks[0].number == num_0; blocks[1].number == num_0-1; ...; blocks[n-1].number == num_0-(m-1)`. This order is not assumed, but validated within the operation.

However, this check does not seem to be enforced in the current implementation. Specifically, no checks are performed in the different block processing functions in *ledger/src/signer/src/bc_advance.c* to ensure the block numbers are sequential. This was also confirmed by modifying the testing data in the file *14-advance-brothers.json*.

Additionally, note that this observation also applies to the “Updating the known ancestor block” use-case of the powHSM, implemented in the *bc_ancestor.c* source file (see *blockchain-bookkeeping.md*):

> As with the previous algorithm (update of the HSM state), the blocks must be provided in newest-to-oldest order.

Recommendation

Add an additional check in the `bc_advance()` function to ensure that the block number of the current block being processed is strictly smaller than the previous block (provided blocks are handled from newest to oldest). Ensure this check is also performed in the `bc_upd_ancestor()` function.

Location

- *ledger/src/signer/src/bc_advance.c*
- *ledger/src/signer/src/bc_ancestor.c*

Retest Results

2022-09-09 – Fixed

Release 3.0.1 (and more specifically commit 986b967) updates the wording of the *blockchain-bookkeeping.md* document to refer to the validity of the chain of parent hashes as opposed to the block numbers, as follows:

> This `blocks` array is indexed from `0` to `m-1` and blocks must be ordered from newest to oldest, i.e. `blocks[0].parent_hash == hash(blocks[1]); blocks[1].parent_hash == hash(blocks[2]); ...; blocks[m-2].parent_hash == hash(blocks[m-1])`. This order is not assumed, but validated within the operation.

This change aligns the documentation with the operations performed in the code. Hence, the finding is considered fixed.

Low

Failure to Validate Signer Authorizer Array Size May Lead to Out-of-Bound Memory Access

Overall Risk	Low	**Finding ID**	NCC-E003945-W4M
Impact	Low	**Component**	UI
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

If more than `MAX_AUTHORIZERS` = 10 distinct authorizers are provided, then the `Signer` authorization process may segfault or behave incorrectly due to out-of-bound memory access.

Description

The list of authorized signers is provided during the initial build by defining the array `AUTHORIZERS_PUBKEYS`, with the threshold hardcoded to a simple majority of the provided signers. For testing purposes, three default keys are provided in *testing.h*, although up to 10 can be provided; see *signer_authorization.h*:

```
// Maximum number of authorizers (increase this if using a greater number)
#define MAX_AUTHORIZERS 10
```

The above value is used to denote the maximum number of authorizers, although it should be noted that **this maximum is not explicitly checked anywhere in the code**. The only usage appears to be in allocating storage to track how many distinct signers have contributed to the threshold, also defined in *signer_authorization.h*:

```
bool authorized_signer_verified[MAX_AUTHORIZERS];
```

As part of the signing process in `do_authorize_signer()` in *signer_authorization.c* the above array is written to.

```
for (int i = 0; i < TOTAL_AUTHORIZERS && !signature_valid; i++) {

        <load signer key, attempt to verify signature>

    // Found a valid signature?
    if (signature_valid) {
        sigaut_ctx->authorized_signer_verified[i] = true;
    }
```

Observe that the index `i` is bounded by `TOTAL_AUTHORIZERS` (the size of the provided authorized signers array) and not by `MAX_AUTHORIZERS`, the size of the array being written to. Therefore, an out-of-bound memory write may occur if too many authorizers are provided.

Note that the process of establishing the set of authorizers is not currently defined and was out of scope for the purposes of this review. Despite this, it is likely to be a highly-controlled and supervised process, with the likelihood of this bug being triggered being exceptionally low. Nevertheless, it represents a missing error check that could be caught and responded to, and it is therefore recommended that the above potential out-of-bounds access be explicitly disallowed.

The function `init_signer_authorization()` is called at initialization, and is responsible for establishing the initial `Signer` hash and iteration provided at build time. This function could also be made responsible for enforcing that `TOTAL_AUTHORIZERS <= MAX_AUTHORIZERS`.

Recommendation

1. Add a check to `init_signer_authorization()` to fail if `TOTAL_AUTHORIZERS` > `MAX_AUTHORIZERS`; or
2. Cap indexes at `MAX_AUTHORIZERS` (e.g., silently ignore any authorizers beyond the maximum number supported).

The former would be a more conservative approach, and would ensure that an incorrect configuration of authorizers is detected.

Location

- *signer_authorization.h*
- *signer_authorization.c*

Retest Results

2022-09-08 – Fixed

Release 3.0.1 adds the following compile time check in *signer_authorization.c*:

```
void init_signer_authorization() {
    // Build should fail when more authorizers than supported are provided
    COMPILE_TIME_ASSERT(TOTAL_AUTHORIZERS <= MAX_AUTHORIZERS);
```

This change effectively prevents the potential out-of-bounds memory access and fixes the issue.

Low

Onboarding State May Not Be Correctly Tracked

Overall Risk	Low	**Finding ID**	NCC-E003945-7DB
Impact	Undetermined	**Component**	UI, tcpsigner
Exploitability	Undetermined	**Category**	Authentication
		Status	Fixed

Impact

The handler for the `RSK_IS_ONBOARD` APDU always returns true, rather than using the `N_onboarded_ui` flag in flash memory. Furthermore, the `N_onboarded_ui` flag may not be set to false when wiping the device, such that the flag will remain set even if the `RSK_WIPE` process fails.

Description

It was observed in *bolos_ux.c* that the onboarding process sets a value in flash memory as part of `RSK_WIPE`, which handles wiping and onboarding a device:

```
    // Turn the onboarding flag on to mark onboarding
    // has been done using the UI
    aux = 1;
    nvm_write((void *)PIC(N_onboarded_ui), &aux, sizeof(aux));
```

This value is referenced in *attestation.c* to validate that the device has been onboarded:

```
    // Verify that the device has been onboarded
    unsigned char onboarded = *((unsigned char*)PIC(N_onboarded_ui));
    if (!onboarded) {
        THROW(ATT_NO_ONBOARD);
        return 0;
    }
```

However, the handler for `RSK_IS_ONBOARD` does not use this value to determine if onboarding via the `UI` is complete:

```
  case RSK_IS_ONBOARD: // Wheter it's onboarded or not
                  reset_if_starting(RSK_IS_ONBOARD);
        uint8_t output_index = CMDPOS;
        SET_APDU_AT(output_index++, os_perso_isonboarded());
        SET_APDU_AT(output_index++, VERSION_MAJOR);
        SET_APDU_AT(output_index++, VERSION_MINOR);
        SET_APDU_AT(output_index++, VERSION_PATCH);
        tx = 5;
        THROW(0x9000);
```

This handler always returns a positive result, and does not actually validate that onboarding has taken place; see *tcpsigner/os.c*

```
#define OS_PERSO_ISONBOARDED_YES 1

unsigned int os_perso_isonboarded() {
    return OS_PERSO_ISONBOARDED_YES;
}
```

Note that this function is defined in `tcpsigner` and not in the `UI` component. It is recommended that this be reviewed by the IOV team to ensure it is the correct behavior, and to update this handler to use the flag stored in flash memory if appropriate, such that all onboarding checks are consistent.

In addition to the above, it was also observed that the handler for `RSK_WIPE` does not explicitly set `N_onboarded_ui` to false at any point. The handler does call `os_perso_wipe()`, but it could not be validated if this operation wipes flash memory, or just secrets held by the secure element. The Ledger Blue SDK contains the following in *os.h*:

```
// any application can wipe the global pin, global seed, user's keys
// disabled for security reasons // SYSCALL void os_perso_wipe(void);
// erase seed, settings AND applications
SYSCALL void os_perso_erase_all(void);
```

Based on the above, it was unclear if `os_perso_wipe()` (or `os_perso_erase_all()`) will correctly wipe flash memory, which would clear the `N_onboarded_ui` flag. If the following are true, it is possible for this flag to be in an incorrect state:

1. A user has previously onboarded via the UI,
2. Flash memory is not zeroed out on wipe,
3. A user onboards again, and the process fails (e.g., an exception is thrown).

In this situation `N_onboarded_ui` will remain set to true, even if onboarding fails. Because this behavior could not be explicitly verified, it is being highlighted for consideration by the IOV team.

Recommendation

1. Update the `RSK_IS_ONBOARD` handler to use the correct `N_onboarded_ui` flag.
2. Ensure that the `N_onboarded_ui` flag is correctly wiped alongside other sensitive information at the start of the onboarding process.

Retest Results

2022-09-08 – Fixed

Regarding the first recommendation, RSK Labs clarified that the highlighted code in the `tcpsigner` component is solely for testing and was not considered in scope for this review. The relevant syscalls are correctly handled by the Ledger SDK when on the device. Therefore, the first half of this finding may be considered a false positive.

Regarding the second recommendation, the handler for `RSK_WIPE` has been updated to explicitly wipe the `N_onboarded_ui` value in flash memory; see *bolos_ux.c*:

```
    case RSK_WIPE: //--- wipe and onboard device ---
        reset_if_starting(RSK_META_CMD_UIOP);

        // Reset the onboarding flag to mark onboarding
        // hasn't been done just in case something fails
        aux = 0;
        nvm_write(
           (void *)PIC(N_onboarded_ui), (void *)&aux, sizeof(aux));
```

As a result, this finding is considered fixed.

Client Response

The TCPSigner is an x86 application for testing, there is no flash memory, and onboarding is not part of this testing component.

Regarding the mock `os_perso_isonboarded()` function, this is done to allow testing under the TCPSigner. In the actual UI, this function is part of the Nano S SDK and its implementation and semantics are the expected ones.

Info

Flash Memory Endurance Considerations

Overall Risk	Informational	**Finding ID**	NCC-E003945-4GK
Impact	Medium	**Component**	signer
Exploitability	Undetermined	**Category**	Other
		Status	Fixed

Impact

Reaching the maximum limit of erase / write cycles to the flash memory may render the Ledger hardware useless, thus preventing correct operation of the powHSM solution.

Description

The powHSM implementation regularly writes to the flash memory of the Ledger device, for example when advancing the blockchain state. These write operations are performed through the preprocessor macros `NVM_RESET()` and `NVM_WRITE()` defined in *ledger/src/signer/src/nvm.h*, both of which call the underlying `nvm_write()` function.

```
#include "os.h"
#define NVM_RESET(dst, size) nvm_write((void*)(dst), NULL, size)
#define NVM_WRITE(dst, src, size) nvm_write((void*)(dst), (void*)(src), size)
#endif
```

The Ledger documentation has specific recommendations and caveats surrounding the use of the flash memory of the Ledger[2].

> The flash memory for the ST31G480, which is the Secure Element used in the Ledger Blue, is rated for 500 000 erase / write cycles. This should be more than enough to last the expected lifetime of the device, but only if applications use it properly. Applications should avoid erasures as much as possible. Here are some techniques for avoiding wearing out the device's flash memory.

Given the frequency at which the powHSM implementation may write to the Ledger flash memory, the total number of erase/write cycles may be reached within the lifetime of a device. Hence, this finding should not only be seen as a warning, but also as a prompt to initiate and introduce development practices to limit writing to flash as much as possible.

Recommendation

The Ledger documentation provides the following recommendations to increase the lifetime of devices.

> consider caching the data in RAM and then flushing to flash memory when the application has finished its operation

> developers should be aware that flash memory pages are aligned to 64-byte boundaries. (...) write amplification can be avoided by making sure that 32 bytes of data is contained entirely within a single page

Consider revisiting the code base with these two recommendations in mind.

2. https://github.com/LedgerHQ/ledger-dev-doc/blob/master/source/userspace/memory.rst#flash-memory-endurance

Retest Results

2022-09-09 – Fixed

Release 3.0.1 (and more specifically commit ca4f306) introduces a number of updates addressing the concerns outlined in this finding. In particular:

- The `initialized` NVM variable was moved into the `N_bc_state` struct to ensure 64-byte alignment.
- A number of `NVM_WRITE` operations were replaced by calls to `HSTORE`, the latter resolving to the standard `memcpy()`, thereby reducing the amount of erase / write cycles to the flash memory.
- One instance of a `NVM_RESET` was replaced by a direct call to `memset`, additionally reducing the amount of erase / write cycles.

These changes align with the recommendations provided above, thereby addressing this finding.

5 Finding Field Definitions

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

Risk Scale

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

Overall Risk

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

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

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

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

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

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

Rating	Description
	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

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

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

6 Review Goals

This section summarizes the review goals initially outlined by the IOV team and provides an overview of how each goal was assessed during the engagement. The following table of contents may be used to jump directly to a given section:

1. Seed cannot be extracted from the device
2. Signature operation authorization cannot be bypassed
   a. Signer update authorization
   b. Transaction signing
3. Recovery mode cannot be accessed without wiping the device first
4. An arbitrary app cannot be successfully used without wiping the device first
5. Arbitrary BIP32 paths cannot be used either for signing or extracting the public key
6. Blockchain state cannot be manipulated without the corresponding PoW

Seed cannot be extracted from the device

The core functionality of the Ledger device is the generation and protection of secret data within a secure element. An overview of this process can be seen in *bolos_ux.c* in the handler for `RSK_WIPE`, which re-establishes the seed once a device has been wiped. This results in a hardware-backed seed with the resulting BIP39 mnemonic generated. During the assessment, no custom code that modifies this process to potentially expose the seed was identified, therefore, the seed on a device running the `UI` and `Signer` apps should be as secure as a standard vanilla Ledger Nano S device.

Per Ledger's documentation:

> It is extremely unlikely for the Device private key to become compromised, because the Secure Element is designed to be a stronghold against such physical attacks. It is theoretically possible to extract the private key, but only with great expense and time, so only an organization such as the NSA could do it.

Note that the above refers to the device private key, and not the seed established during setup. Indeed, the security guidelines for app developers explicitly acknowledges that a poorly coded application could leak secrets. For examples, the guidance of "Restrict Apps to Coin-Specific BIP32 Prefix" is given, as well as "Do not allow the host to freely manipulate key-pairs". Unsafe interfaces may allow a user to perform mathematical operations using a private key that would cause its value to leak.

The Engagement Notes section surveys the Ledger Security Guidelines and did not find any violations or unsafe operations that would allow secret information to be recovered. Similarly, later in this section the requirement that "Arbitrary BIP32 paths cannot be used" is validated, which further reinforces that keys cannot be revealed mistakenly. As a final observation, deterministic ECDSA signatures (RFC6979) are leveraged, which prevents a poor source of randomness from leaking signing keys.

Known Vulnerabilities

For completeness, this sub-section presents a known potential attack on a Ledger device. It should be noted that this attack **does not** affect the `powHSM` application, as the only practical means of exploit involve interaction with the UI.

The Ledger Nano S consists of two distinct ST microprocessors:

1. An STM32F042K Microcontroller
2. An SST31H320 Secure Microcontroller (the "secure element")

The secure element handles the storage and management of cryptographic keys, validation of the bootloader, and provides some hardware-based random number generation and implementation of cryptographic primitives. The other MCU manages the peripherals (e.g., USB and buttons) and notifies the secure element whenever new data is ready to be received. Applications are executed entirely in the secure element.

The so-called “F00DBABE” vulnerability (YouTube Link[3]) is a mechanism for injecting an untrusted firmware onto the device. While this does not directly compromise secrets stored within the secure element, it does give an attacker the ability to run arbitrary code in the bootloader, which includes altering the contents of the screen, and emulating button presses.

The Ledger bootloader uses a “magic value” of `0xF00DBABE` at a specific address to manage firmware updates. When the firmware is updated, this magic value is cleared, and the bootloader prevents any non-authorized firmware from writing to this memory location. Therefore, if an attacker modifies the firmware, the value will be cleared and the firmware will not be executed. If the firmware is installed and attestation succeeds, then the validated firmware can re-write this magic value and the device will continue to load the newly-installed firmware.

The “vulnerability” in this case is a hardware configuration which allows two regions of virtual memory to map to the same physical location in flash memory. Therefore, even though unauthorized writes are prevented by the firmware, a second unprotected virtual memory address exists pointing to the same physical address such that untrusted firmware can write the `0xF00DBABE` value without passing attestation. An attacker cannot use custom firmware to extract the seed, but they could potentially alter the behavior of the UI in a way that misleads a user into signing something they did not intend to.

Note that this attack requires physical access to the device and the ability to influence user behavior. Such an attacker could potentially leverage other surveillance techniques to learn the user’s PIN and use the device directly without further compromise. But such an attacker would still not gain access to the seed, unless they can break the security of the underlying secure element. Because the `powHSM` application does not involve active interaction with the UI, this vulnerability does not have practical implications for `powHSM`’s use case.

Limitations

While physical devices were provided as part of this assessment, a review of the physical security of the Ledger Nano S and the included secure element was not in scope. This includes the investigation of side channel attacks, or attempts to reverse engineer the behavior of the secure element. The device was treated as a black box and was interacted with using the same interfaces a regular user would (e.g., via USB). A hardware-focused penetration test may yield findings and observations not present in this report.

Re-test Results

None.

Signature operation authorization cannot be bypassed

Signature operation authorization may refer to two processes within the codebase:

1. The process by which the `UI` component authorizes the updating of the `Signer` component, as documented at: https://github.com/rsksmart/rsk-powhsm/blob/3.0.0/docs/signer-authorization.md.
2. The process by which the `Signer` component facilitates the signing of a transaction.

3. https://www.youtube.com/watch?v=nNBktKw9Is4

This section will focus on the first item. The process of signing a transaction is covered later in part of Arbitrary BIP32 paths cannot be used either for signing or extracting the public key and Transaction signature operation authorization cannot be bypassed.

The `Signer` authorization process is handled via the main loop in *bolos_ux.c*:

```
        case INS_SIGNER_AUTHORIZATION:
            reset_if_starting(INS_SIGNER_AUTHORIZATION);
            tx = do_authorize_signer(rx, &sigaut_ctx);
            THROW(0x9000);
            break;
```

The core logic handled by `do_authorize_signer()` in *signer_authorization.c*. This function handles number of APDU operations, with the primary two being:

- `SIG_AUT_OP_SIGVER`: Read in a new `Signer` hash and iteration to be verified.
  - The hash and iteration are written to the current context.
  - The iteration count is checked against the current iteration.
  - The EIP-712 to-be-signed hash is calculated and stored in the context.
- `SIG_AUT_OP_SIGN`: Process a signature on the `Signer` hash for threshold authorization.
  - Receives a signature and checks it in sequence against the list of known authorized signers.
  - Checks are performed in such a way that multiple signatures from the same signer will not count as additional signatures towards the threshold.
  - If no valid authorized signer is found, an error is returned. Note that finding "Inconsistent Threshold Signature Validation Criteria" elaborates on the implemented validation criteria (e.g., fail vs proceed on error).
  - If the threshold has been met, the stored `Signer` hash and iteration are updated.
  - If the threshold has not been met, `SIG_AUT_OP_SIGN_RES_MORE` is returned. The current progress remains tracked via the context.

The remaining operations simply fetch state and do not alter the authorization context:

- `SIG_AUT_OP_GET_CURRENT`: Return the current `Signer` hash and iteration.
- `SIG_AUT_OP_GET_AUTH_COUNT`: Return the total number of authorizers.
- `SIG_AUT_OP_GET_AUTH_AT`: Return the public key of the authorizer at a given index.

The list of authorized signers is provided during the initial build by defining the array `AUTHORIZERS_PUBKEYS`, with the threshold hardcoded to a simple majority of signers. For testing purposes, three default keys are provided in *testing.h*, although up to 10 can be provided; see *signer_authorization.h*:

```
// Maximum number of authorizers (increase this if using a greater number)
#define MAX_AUTHORIZERS 10
```

The above value is used to denote the maximum number of authorizers, although it should be noted that **this maximum is not explicitly checked anywhere in the code**. Finding "Failure to Validate Signer Authorizer Array Size May Lead to Out-of-Bound Memory Access" details a potential out-of-bound memory access relating to this missing check.

A pre-requisite to the `Signer` authorization process is a call to `init_signer_authorization()`, which occurs in *bolos_ux.c*. If the state is not initialized, this function loads the currently stored signer hash and currently stored signer iteration.

The existence of a valid `Signer` hash is enforced by the device in *bolos_ux.c* by calling the function `is_authorized_signer()`, which compares the provided app hash to the authorized `Signer` hash:

```c
/*
 * Tell whether the given signer hash is authorized to run
 * as per the current signer authorization status.
 *
 * @arg[in] signer_hash     the signer hash
 */
bool is_authorized_signer(unsigned char* signer_hash) {
    return !memcmp(N_current_signer_status.signer.hash,
                   signer_hash,
                   sizeof(N_current_signer_status.signer.hash));
}
```

The process of validating and enforcing the correct `Signer` hash is reviewed in the later section An arbitrary app cannot be successfully used without wiping the device first.

Based on the above, it appears that the requirements set forth in *signer-authorization.md* are realized in the current implementation.

As a final consideration, the following notes from the documentation are of particular importance:

> With the current implementation, the M keypairs in the N of M authorization scheme are fixed. An improvement worth considering is implementing an operation that allows for the updating of said keypairs.

> The potential implementations and consequent security implications of this and other proposed changes should be analysed carefully before actually moving forwards.

The establishment and management of authorizers was outside the scope of this review. The security of the entire process is reliant on the correct instantiation and behavior of the authorizers, and there do not appear to be any documented processes related to the management of authorizers over time.

Re-test Results

See finding "Inconsistent Threshold Signature Validation Criteria" for additional details.

Recovery mode cannot be accessed without wiping the device first

The primary mechanism used to prevent recovery mode from being accessed is described in *attestation.md*:

> At onboarding, the user-entered pin is required to contain at least one non-numeric character, and the recovery screen for the Ledger device only allows for the manual input of a fully numeric pin. This in turn implies that the only way of accessing the recovery screen after the UI is installed and the device is onboarded is by entering an invalid pin three times, which would wipe the device - including any generated keys.

Recovery mode is enabled by the bootloader when the Ledger device is powered on with the right button held down. As noted above, the user will be prompted for their PIN, if one is set, prior to being granted access to recovery mode.

When initially setting up the Ledger device from a blank state, the user is prompted to select a PIN consisting of 4-8 numerical digits, selected via the device UI. Overall, there are four points within the code where a user's PIN can be set:

1. During the initial device setup,
2. When re-initializing a device from a passphrase,
3. In the `RSK_WIPE` handler ("wipe and onboard device"),
4. In the `RSK_NEWPIN` handler.

The first two cases are part of the regular Ledger setup, and the latter two are part of the `UI` component in *bolos_ux.c*.

The handler for `RSK_WIPE` performs the following:

```
    case RSK_WIPE: //--- wipe and onboard device ---
                   reset_if_starting(RSK_META_CMD_UIOP);
#ifndef DEBUG_BUILD
                   validate_pin(G_bolos_ux_context.pin_buffer);
#endif
                   // Wipe device
                   os_global_pin_invalidate();
                   os_perso_wipe();

<...snip...>

                  // Set PIN
                   os_perso_set_pin(
                       0,
                       (unsigned char *)G_bolos_ux_context.pin_buffer + 1,
                       G_bolos_ux_context.pin_buffer[0]);
                   // Finalize onboarding
                   os_perso_finalize();
                   os_global_pin_invalidate();
                   SET_APDU_AT(1, 2);
                   SET_APDU_AT(
                       2,
                       os_global_pin_check(
                           (unsigned char *)G_bolos_ux_context.pin_buffer + 1,
                           G_bolos_ux_context.pin_buffer[0]));
                    // Clear pin buffer
                   explicit_bzero(G_bolos_ux_context.pin_buffer,
                                  sizeof(G_bolos_ux_context.pin_buffer));
<...snip...>
```

From the above code, wiping the device first calls `validate_pin()` on `G_bolos_ux_context.pin_buffer`:

```
/*
 * Do pin validations on the given pin buffer
 * If pin validations fail, throw
 */
void validate_pin(char *pin_buffer) {
    // Check PIN length
    if (pin_buffer[0] != MAX_PIN_LENGTH) {
        THROW(0x69a0);
    }
    // Check if PIN is alphanumeric
    int isAlphanumeric = 0;
    for (int i = 0; i < MAX_PIN_LENGTH; i++) {
        if (pin_buffer[i + 1] > '9') {
            isAlphanumeric = 1;
        }
    }
    if (!isAlphanumeric) {
        THROW(0x69a0);
    }
}
```

This function ensures that at least one character within `pin_buffer` (populated in the `RSK_PIN_CMD` handler) is non-numeric. The device is wiped and a new seed is generated, and then the PIN in the `pin_buffer` is restored via `os_perso_set_pin()`. Next, the existing PIN-based authentication is invalidated (i.e., PIN must be provided again for sensitive operations), and the result of `os_global_pin_check()` (i.e., re-validate the PIN) is written to the returned APDU as the result. The in-memory `pin_buffer` is then cleared.

Note that this handler is the one and only place in the code where the flag `N_onboarded_ui` is set, specifying that the UI has been correctly onboarded. Therefore, if this flag is set, then the device should be onboarded with a PIN that contains non-numeric characters. The finding "Onboarding State May Not Be Correctly Tracked" details some additional concerns around the use and management of this flag.

The fourth case where the PIN is set is in the handler for `RSK_NEWPIN`:

```
    case RSK_NEWPIN:
        reset_if_starting(RSK_META_CMD_UIOP);
#ifndef DEBUG_BUILD
        validate_pin(G_bolos_ux_context.pin_buffer);
#endif
        // Set PIN
        os_perso_set_pin(
            0,
            (unsigned char *)G_bolos_ux_context.pin_buffer + 1,
            G_bolos_ux_context.pin_buffer[0]);
        // check PIN
        os_global_pin_invalidate();
        SET_APDU_AT(1, 2);
        SET_APDU_AT(
            2,
            os_global_pin_check(
                (unsigned char *)G_bolos_ux_context.pin_buffer + 1,
                G_bolos_ux_context.pin_buffer[0]));
        tx = 3;
        THROW(0x9000);
        break;
```

This follows the same logic as in the `RSK_WIPE` case, where the PIN is validated (i.e., is the correct length and contains at least one non-numeric character), invalidated, and the result of `os_global_pin_check()` is returned. The function does differ from the `RSK_WIPE` case in that **the `pin_buffer` is not wiped. If there is no use case that requires this value to remain**

in memory, it would be prudent to explicitly delete it. It should also be noted that the PIN is potentially cached in one other location:

```
    case RSK_PIN_CMD: // Send pin_buffer
        reset_if_starting(RSK_META_CMD_UIOP);
        pin = APDU_AT(2);
        if ((pin >= 0) && (pin <= MAX_PIN_LENGTH)) {
            G_bolos_ux_context.pin_buffer[pin] = APDU_AT(3);
            // We don't need the prepended length for the pin
            // cache, so it is one byte smaller
            if (pin < sizeof(G_pin_cache) - 1) {
                G_pin_cache[pin] = APDU_AT(3);
                G_pin_cache[pin + 1] = 0;
            }
        }
        THROW(0x9000);
        break;
```

This value is checked on line 970 as part of `BOLOS_UX_CONSENT_APP_ADD`, and is not explicitly deleted at any point in the UI. **It is recommended to review the usage and deletion of `G_pin_cache` and `pin_buffer`, and to potentially combine them to ensure the PIN is managed in memory as intended.**

Based on the above observations, it appears that a device that has been successfully onboarded with the UI will always contain a maximum-length PIN with at least one non-digit character, thereby preventing the entering of the correct PIN via the device UI. The only identified mechanism to specify such a PIN is via the `RSK_PIN_CMD` handler via an incoming APDU. No other mechanisms to bypass this process were identified.

Section Engagement Notes, subsection *Unexpected UI State*, identified unexpected default Ledger behavior around recovery mode.

Re-test Notes:
As part of release 3.0.1, commit 881627a unified the PIN buffers and replaced the `validate_pin()` function with a new function `is_pin_valid()` in *pin.c*, which performs a similar check to ensure the PIN contains at least one alphabetic character.

Per the recommendation above, the handler for `RSK_NEWPIN` now explicitly clears the PIN buffer, aligning it with the `RSK_WIPE` case; see *bolos_ux.c*:

```
    // Clear pin buffer
    explicit_bzero(G_pin_buffer, sizeof(G_pin_buffer));
    THROW(APDU_OK);
```

This patch effectively addresses the recommendations highlighted in this sub-section.

An arbitrary app cannot be successfully used without wiping the device first

Recall from earlier that the correctness of the `Signer` authorization process was established. The function `is_authorized_signer()` is defined in *signer_authorization.c*:

```
/*
 * Tell whether the given signer hash is authorized to run
 * as per the current signer authorization status.
 *
 * @arg[in] signer_hash     the signer hash
 */
```

```
bool is_authorized_signer(unsigned char* signer_hash) {
    return !memcmp(N_current_signer_status.signer.hash,
                   signer_hash,
                   sizeof(N_current_signer_status.signer.hash));
}
```

This function returns true if and only if the currently stored `Signer` hash matches the provided hash. It is relied upon both when an app is added to the Ledger device, or by the function `is_app_version_allowed()` in *bolos_ux.c*:

```
// Check if we allow this version of the app to execute.
int is_app_version_allowed(application_t *app) {
    if (is_authorized_signer(app->hash))
        return 1;
    return 0;
}
```

This function is then leveraged by `run_first_app()` to load the first non-UI app, as long as it matches the allowed `Signer` version:

```
// run the first non ux application
void run_first_app(void) {
    unsigned int i = 0;
    while (i < os_registry_count()) {
        application_t app;
        os_registry_get(i, &app);
        if (!(app.flags & APPLICATION_FLAG_BOLOS_UX)) {
            if (is_app_version_allowed(&app)) {
                G_bolos_ux_context.app_auto_started = 1;
                screen_stack_pop();
                io_seproxyhal_disable_io();
                os_sched_exec(i); // no return
            }
        }
        i++;
    }
}
```

In other words, if the `run_first_app()` function results in an app being launched, it should be the correct `Signer` app.

Automatically Launching the `Signer`

The primary interface of the Ledger UI is the dashboard. The handler for the dashboard event calls `run_first_app()` when `autoexec` is set:

```
        case BOLOS_UX_DASHBOARD:
            screen_wake_up();

            // apply settings when redisplaying dashboard
            screen_settings_apply();

            // when returning from application, the ticker could have been
            // disabled
            io_seproxyhal_setup_ticker(100);
            // Run first application once

            if (autoexec) {
                autoexec = 0;
                run_first_app();
            }
            screen_dashboard_init();
            break;
```

This `autoexec` variable is set to `1` whenever a command completes normally:

```
    case RSK_END_CMD: // return to dashboard
        reset_if_starting(RSK_END_CMD);
        autoexec = 1;
        goto return_to_dashboard;
```

Therefore, the expected behavior when returning to the dashboard is an automatic launch of the `Signer` app. The app is also launched as part of the handler for `BOLOS_UX_CONSENT_GET_DEVICE_NAME`:

```
    case BOLOS_UX_CONSENT_GET_DEVICE_NAME:
        screen_wake_up();
        // GET_DEVICE_NAME event override to reload app
        run_first_app();
        // screen_consent_get_device_name_init();
        break;
```

This appears to be a method of force-loading the app by hijacking a common operation. It was not immediately obvious what triggers this event, but it is likely to be a common operation, perhaps performed as part of a normal startup.

Manually Launching the `Signer`

In *bolos_ux_dashboard.c*, there exists logic to handle a left+right button release (e.g., to perform the selected action). This contains notes of a workaround based on apps persisting after a PIN reset:

```
    // if application is not signed when installed, nor is using the
    // issuer key,
    // then ask if the user really is ok to run it, this solves the
    // security flaw
    // that apps are not wiped when seed is wiped after 3 wrong PIN
    // attempts.

    <...snip...>

  // requested non genuine validation
    if ((db.app.flags &
         (APPLICATION_FLAG_ISSUER | APPLICATION_FLAG_CUSTOM_CA |
         APPLICATION_FLAG_SIGNED)) == 0) {

        <...snip...>

         // override the consent callback, just use the logic
         G_bolos_ux_context.screen_stack[0].button_push_callback =
              screen_dashboard_unsigned_app_button;

        <...snip...>

        } else {
           // delegate boot
           if (is_app_version_allowed(&db.app)) {
               screen_dashboard_disable_bolos_before_app();
               os_sched_exec(db.os_index); // no return
           }
        }
```

The highlighted conditional above is true if the app is from an untrusted issuer, is unsigned, or a custom CA is being used. If any of these are true, then the code eventually hits the custom callback `screen_dashboard_unsigned_app_button()`. Otherwise, the default case proceeds to check the app version and continues. The code within the custom callback applies similar logic to the default case, if and only if the `DEBUG_BUILD` flag is set, meaning the command will succeed for a non-genuine validation only in a debug build.

New App Installation

In the case of adding a new app to the device, the handler in `bolos_ux_main()` will hit the following:

```
case BOLOS_UX_CONSENT_APP_ADD:
    if (is_authorized_signer(
            G_bolos_ux_context.parameters.u.appadd.appentry.hash)) {
        // PIN is invalidated so we must check it again
        os_global_pin_check(G_pin_cache, strlen(G_pin_cache));
        G_bolos_ux_context.exit_code = BOLOS_UX_OK;
        break;
    } else {
        G_bolos_ux_context.exit_code = BOLOS_UX_CANCEL;
    }

    screen_wake_up();
    break;
```

As can be seen, the operation is canceled if the `Signer` is not correctly authorized.

Summary

Based on the above observations, and the earlier review of the `Signer` authorization process, the UI appears to enforce that only an authorized `Signer` app can be installed, and that the `Signer` app should be automatically launched in normal circumstances. If a user attempts to manually launch an app (e.g., after a PIN reset), then the same authorization check will be enforced via `is_app_version_allowed()`.

Additional investigation may be needed to ensure that the above cases are comprehensive, but it appears that once a device is correctly initialized, only the `Signer` app may be launched. Unless, of course, the device is factory-reset.

Re-test Results

None.

Arbitrary BIP32 paths cannot be used for signing or extracting the public key)

Note that this section covers both the validation of BIP32 paths, as well as the resulting use of the derived keys.

Let us first consider the case of extracting the public key. The relevant code path is located inside the file *ledger/src/signer/src/hsm.c*, where a `switch` statement defines the case `INS_GET_PUBLIC_KEY` on line 111. Inside this case, the conditional statement highlighted below ensures only hardcoded paths may be used.

```
// Derives and returns the corresponding public key for the given path
case INS_GET_PUBLIC_KEY:

  // <snip>

  if (!(pathRequireAuth(APDU_DATA_PTR - 1) ||
        pathDontRequireAuth(APDU_DATA_PTR - 1))) {
      // If no path match, then bail out
      THROW(0x6A8F); // Invalid Key Path
  }
```

Specifically, this statement ensures that the path provided in the APDU is either contained in the static `authPaths` array or in the static `noAuthPaths` array, defined in the *ledger/src/signer/src/pathAuth.c* file, by calling the functions `pathRequireAuth()` and `pathDontRequireAuth()`.

The signing use-case is a little more complicated. First, the *hsm.c* file dispatches the `auth_sign()` function upon notification of an `INS_SIGN` operation:

```
    case INS_SIGN:
        reset_if_starting(INS_SIGN);
        tx = auth_sign(rx);
        break;
```

The `auth_sign()` function (located in *ledger/src/signer/src/auth.c*) performs different function calls depending on the current state of the powHSM and on the type of path to be used for derivation. The relevant code excerpt is provided below, for reference.

```
unsigned int auth_sign(volatile unsigned int rx) {
    unsigned int tx;

    // Check we receive the amount of bytes we requested
    // (this is an extra check on the legacy protocol, not
    // really adding much validation)
    if (auth.state != AUTH_ST_START && auth.state != AUTH_ST_MERKLEPROOF &&
        APDU_DATA_SIZE(rx) != auth.expected_bytes)
        THROW(0x6A87);

    switch (APDU_OP() & 0xF) {
    case P1_PATH:
        if ((tx = auth_sign_handle_path(rx)) == 0)
           break;
        return tx;
    case P1_BTC:
        return auth_sign_handle_btctx(rx);
    case P1_RECEIPT:
        return auth_sign_handle_receipt(rx);
    case P1_MERKLEPROOF:
        if ((tx = auth_sign_handle_merkleproof(rx)) == 0)
           break;
        return tx;
    default:
        // Invalid OP
        THROW(0x6A87);
    }

    if (auth.state != AUTH_ST_SIGN)
        THROW(0x6A89); // Invalid state

    tx = do_sign(auth.path,
                 RSK_PATH_LEN,
                 auth.sig_hash,
                 sizeof(auth.sig_hash),
                 APDU_DATA_PTR,
                 APDU_TOTAL_DATA_SIZE);
```

The critical `do_sign()` operation may only be called in two instances, namely after the execution hits one of the two `break` statements in the `switch` operator above, and only when the state is `AUTH_ST_SIGN`.

There are two different paths that can lead to successful execution of the `do_sign()` function, depending on whether the path requires authorization of not. Consider the execution flow of a *sign* operation, starting with a call to the `auth_sign_handle_path()` function. This function starts by checking that the current state is `AUTH_ST_PATH` and throws an exception otherwise. After that, the execution checks whether the path requires authorization by calling the `pathRequireAuth()` function, and if so, transitions to the state `AUTH_ST_BTCTX`. Alternatively, if the path does not require authorization (checked with the function call `pathDontRequireAuth()`), the state transitions to `AUTH_ST_SIGN`. In this latter case, the value returned by the `auth_sign_handle_path()` function is `0`, and as such the execution will break out of the `switch` statement above, and proceed to the signing operation. The `pathRequireAuth()` case is a little more complex, and is detailed below.

```
unsigned int auth_sign_handle_path(volatile unsigned int rx) {
  if (auth.state != AUTH_ST_PATH)
      THROW(0x6A89); // Invalid state

  // <snip>

  if (pathRequireAuth(APDU_DATA_PTR)) {

      // <snip>

      auth_transition_to(AUTH_ST_BTCTX);
      return TX_FOR_TXLEN();
  } else if (pathDontRequireAuth(APDU_DATA_PTR)) {

      // <snip>

      auth_transition_to(AUTH_ST_SIGN);
      return 0;
  }
```

Now, if the path does require authorization, the execution proceed through the other functions present in the `switch` statement, in the following order.

1. `auth_sign_handle_btctx()`
2. `auth_sign_handle_receipt()`
3. `auth_sign_handle_merkleproof()`

These three functions all start by checking that the current execution state is consistent with what they expect, specifically:

- `auth_sign_handle_btctx()` checks that the state is `AUTH_ST_BTCTX` and transitions to `AUTH_ST_RECEIPT` upon successful processing.
- `auth_sign_handle_receipt()` checks that the state is `AUTH_ST_RECEIPT` and transitions to `AUTH_ST_MERKLEPROOF` upon successful processing.
- `auth_sign_handle_merkleproof()` checks that the state is `AUTH_ST_MERKLEPROOF` and transitions to `AUTH_ST_SIGN` upon successful processing.

At this stage, the execution may proceed with signing.

Summary

Based on the above observations, the NCC Group team confirmed that the execution is consistent with the following state diagram, with appropriately gated state transitions.

$$\begin{tikzcd}
{} \arrow[r] & \text{AUTH\_ST\_PATH} \arrow[r] \arrow[d] & \text{AUTH\_ST\_BTCTX} \arrow[r] & \text{AUTH\_ST\_RECEIPT} \arrow[dl] \\
& \text{AUTH\_ST\_SIGN} \arrow[d] & \text{AUTH\_ST\_MERKLEPROOF} \arrow[l] & \\
& {} & &
\end{tikzcd}$$

As such, arbitrary BIP32 paths cannot be used for either signing or extracting the public key since successful execution of these operations goes through calls to `pathRequireAuth()` or `pathDontRequireAuth()`, which check that the path provided is contained in the static `authPaths` or `noAuthPaths` arrays. Furthermore, the `do_sign()` function is gated by the strict state transitions depicted above, thereby ensuring that the generation of a signature follows the defined process.

Re-test Notes

None.

Transaction signature operation authorization cannot be bypassed

The previous section presented a walkthrough of the process used to derive and utilize derived keys. As seen in the state diagram presented earlier, the code enforces two potential outcomes based on the behavior of `auth_sign_handle_path()`:

1. If `pathRequireAuth()` returns true, we transition to the `AUTH_ST_BTCX` and must proceed though the remaining states before signing takes place;
2. If `pathDontRequireAuth()` returns true, then we proceed directly to signing.

The resulting signature is computed in *auth.c*:

```
    if (auth.state != AUTH_ST_SIGN)
        THROW(AUTH_ERR_INVALID_STATE); // Invalid state

    tx = do_sign(auth.path,
                 RSK_PATH_LEN,
                 auth.sig_hash,
                 sizeof(auth.sig_hash),
                 APDU_DATA_PTR,
                 APDU_TOTAL_DATA_SIZE_OUT);
```

This represents the only instance within the code where the `do_sign()` function is called. Therefore, a signature will never be generated unless the current state is `AUTH_ST_SIGN`, and this state can only be reached by validly progressing through the state transitions as described earlier.

Re-test Notes

None.

Blockchain state cannot be manipulated without the corresponding PoW

The blockchain state maintained on the Ledger can be updated using functions defined in the *bc_advance.c* and *bc_ancestor.c* source files.

The specific algorithms to perform these state updates are provided in the reference *blockchain-bookkeeping.md* documentation. Specifically, the *Updating* section describes the operation `advanceBlockchain` and `resetAdvanceBlockchain` to update the blockchain state of an initialized powHSM device. This section additionally provides a number of checks and validation rules to ensure that illegitimate state transitions may not occur.

The NCC Group team gathered these validation rules and checked whether they were correctly enforced in the implementation, in order to prevent malicious manipulation of the blockchain state without the corresponding Proof-of-Work. The following list summarizes these validation criteria and provides code pointer

- This blocks array is indexed from `0` to `m-1` and **blocks must be ordered from newest to oldest**, i.e. `blocks[0].number == num_0; blocks[1].number == num_0-1; ...; blocks[n-1].number == num_0-(m-1)`. This order is not assumed, but validated within the operation.
  - This check does not seem to be performed, as also described in finding "Block Number Validation in Blockchain State Update Does Not Match Documentation".
- Blocks must also be valid, i.e. `pow_valid(blocks[i]) == true for 0 < i < m`. This is also validated within the operation.
  - While no function called `pow_valid()` currently exists in the C code base, the `bc_advance()` function seems to perform all the necessary checks on lines 955-957
- For each `0 <= i < n`, with `p` being the length of `brothers[i]`, **it must be the case** that for every `0 <= j < p`,
  - `blocks[i].parent_hash == brothers[i][j].parent_hash`
    - Checking that the blocks are actually brothers is performed in the function `str_end()` on line 619
  - `pow_valid(brothers[i][j]) == true`
    - While no function called `pow_valid()` currently exists in the C code base, the `bc_advance()` function also seems to perform all the necessary checks on lines 955-957.
  - `hash(blocks[i]) <> hash(brothers[i][j])`
    - Ensuring that the block and its brother are different is performed on line 676 in the function `str_end()`

    - `j < (p-1) => hash(brothers[i][j]) < hash(brothers[i][j+1])`.
        - Ensuring that the brothers are sent in ascending hash order is performed on line 683 in the function `str_end()`

  The following code excerpt shows these last two validation checks.

```
    // Check that the brother is not the same as the main block
    if (!PROCESSING_BLOCK() &&
        HEQ(block.main_block_hash, block.block_hash)) {
        FAIL(BROTHER_SAME_AS_BLOCK);
    }

    // Check that the brothers are sent in ascending order
    // wrt their block hash
    if (!PROCESSING_BLOCK() &&
        !HLT(block.prev_brother_hash, block.block_hash)) {
        FAIL(BROTHER_ORDER_INVALID);
    }
```

- Also, it must be the case (although it is also not assumed and therefore validated) that:
    - `blockchain_state.updating.in_progress == false` or,
    - `blockchain_state.updating.in_progress == true` and `hash(blocks[0]) == blockchain_state.updating.next_expected_block`
        - The function `bc_adv_prologue()`, which is called as soon as a whole BTC merge mining header is received, ensures that either `N_bc_state.updating.in_progress` and `block.block_hash == N_bc_state.updating.next_expected_block` are both true, or that `blockchain_state.updating.in_progress` is false, in which case it kicks off the updating process.
- Last but not least, in the last invocation of a single advance operation, it must also be the case (this is also validated before updating the state) that blocks `[m-1].parent_hash == blockchain_state.best_block`, i.e. the last given block header's parent hash must correspond to the best block stored in the current HSM state.
    - This check seems to be correctly performed on line 389 of the function `(bc_mm_header_received)`

```
    } else if (HNEQ(aux_bc_st.prev_parent_hash, block.block_hash)) {
        LOG_HEX("PAR", aux_bc_st.prev_parent_hash, HASH_LEN);
        LOG_HEX("BLK", block.block_hash, HASH_LEN);
        FAIL(CHAIN_MISMATCH);
    }
```

Re-test Notes

None.

7 Engagement Notes

This informal section contains notes and observations generated during the project. There are no security issues that are not already reported in the preceding findings, but the following content may be useful for discussion purposes. This section is not intended to be exhaustive.

General

Error Propagation in Arithmetic Functions

The function `mpAdd()` defined in *ledger/src/signer/src/bigdigits.c* performs multi-precision addition of two big integer values. This function returns a carry if the computation overflowed. Thus, the expected return value is either 0 or 1. However, that function may alternatively return an error, in the form of a `MAX_DIGIT` value, as shown in the code excerpt below.

```
    /*     w can't be the same as v
        Stop if assert is working, else return error (overflow = -1)
    */
    if (w == v) {
        assert(w != v);
        return MAX_DIGIT;
    }
```

Calling functions may not realize that an error scenario occurred. For example, the function `accum_difficulty()` in *ledger/src/signer/src/bc_diff.c* returns that carry value directly, even though its documentation states that it may only return 0 or 1 upon success, as can be seen in the code excerpt below.

```
   * @ret
   *   1 if there's carry
   *   0 if there's no carry
   *   BCDIFF_ERR_INVALID if an error occurs
   */
  DIGIT_T accum_difficulty(DIGIT_T difficulty[], DIGIT_T total_difficulty[]) {
      DIGIT_T aux[BIGINT_LEN];
      DIGIT_T carry = mpAdd(aux, difficulty, total_difficulty, BIGINT_LEN);
      // <snip>

      return carry;
  }
```

This also happens in the function `spDivide()` in *ledger/src/signer/src/bigdigits.c*, which may return an overflow value. This function is not used directly by the IOV code, but it is called within the same file by the function `mpShortDiv()`, again without checking for a potential error in the overflow.

Re-test Notes

As of release 3.0.1, the `accum_difficulty()` function has been updated to return the documented error code.

Hard-Coded Data Type Sizes

Some inconsistencies can be observed in the parameters provided to the different `SAFE_MEMMOVE()` calls; consider for example the following two excerpts where the length parameter provided is computed with a `sizeof()` call in *ledger/src/signer/src/auth_path.c*:

```
    // Read derivation path
    SAFE_MEMMOVE(auth.path,
                 sizeof(auth.path),
                 MEMMOVE_ZERO_OFFSET,
                 APDU_DATA_PTR,
                 APDU_TOTAL_DATA_SIZE,
                 1, // Skip path length (first byte)
                 sizeof(auth.path),
                 THROW(0x6A87));
```

and by using a constant in *ledger/src/signer/src/hsm.c*.

```
        // Derive the public key
        SAFE_MEMMOVE(auth.path,
                     sizeof(auth.path),
                     MEMMOVE_ZERO_OFFSET,
                     APDU_DATA_PTR,
                     APDU_TOTAL_DATA_SIZE,
                     MEMMOVE_ZERO_OFFSET,
                     RSK_PATH_LEN * sizeof(uint32_t),
                     THROW(0x6A8F));
```

Re-test Notes

As of release 3.0.1, the highlighted code block above has been updated to correctly use `sizeof(auth.path)`.

Potentially Incorrect Event Handler

The handler for `BOLOS_UX_BOOT_RECOVERY` has been commented out in *bolos_ux.c*

```
         case BOLOS_UX_BOOT_RECOVERY:
        /*
        screen_boot_recovery_init();
        break;
        */
        case BOLOS_UX_DASHBOARD:
           screen_wake_up();

           // apply settings when redisplaying dashboard
           screen_settings_apply();

           // when returning from application, the ticker could have been
           // disabled
           io_seproxyhal_setup_ticker(100);
           // Run first application once

           if (autoexec) {
               autoexec = 0;
               run_first_app();
           }
           screen_dashboard_init();
           break;
```

Note that **both** the call to `screen_boot_recovery_init()` and the subsequent `break` have been commented out. Therefore, the `switch` statement will fall through to the `BOLOS_UX_DASHBOARD` case. It was not clear if this is intended behavior, as the intention may have been to remove the recovery case rather than to change its behavior. If the intention is a “no op” for the `BOLOS_UX_BOOT_RECOVERY` case, then the code appears incorrect. If the intention is to fall through to the dashboard, then the behavior appears correct. In either case, an additional comment clarifying the behavior would remove any outstanding ambiguity.

Re-test Notes

As of release 3.0.1, the commented code and the line `case BOLOS_UX_BOOT_RECOVERY:` have been deleted, so the code will no longer fall through to a potentially incorrect case.

Code quality and documentation

Notation Deviation from Documentation

The various references define many constants, functions and algorithms that the implementation defines under different names, making the process of ensuring the correctness of algorithms more complex. As an example, consider the constant `MINIMUM_CUMULATIVE_DIFFICULTY` and the functions `hash()` and `pow_valid()` that do not exist as such in the implementation (but are defined in *docs/blockchain-bookkeeping.md*).

Stale Code and Debug Code

Some portions of the code base still have commented code, such as in the *ledger/src/signer/src/keccak256.c* file:

```
        /* apply Keccak rho() transformation */
        for (uint8_t i = 1; i < 25; i++) {
           //state[i] = ROTL64(state[i], pgm_read_byte(&rhoTransforms[i - 1]));
           state[i] = ROTL64(state[i], getConstant(TYPE_RHO_TRANSFORM, i - 1));
        }
```

Similarly, there is a commented `printf()` call in *ledger/src/signer/src/trie.c*:

```
        // printf("st: %u, i: %u/%u, val: 0x%02x\n", ctx->state, i, len,
        // buf[i]);
```

Re-test Notes

As of release 3.0.1, the commented code snippets highlighted above have been deleted.

Stale Links in Documentation

The links to the Ledger documentation in *docs/attestation.md* are dead

> [the ledger documentation](https://ledger.readthedocs.io/en/latest/bolos/features.html#attestation)

They should be replaced by https://developers.ledger.com/docs/nano-app/bolos-features/.

Re-test Notes

As of release 3.0.1, the documentation has been updated with current links.

Incorrect Code Comment

In the file *ledger/src/signer-certificate/src/main.c*, the following comment indicates that a given length should be 32, when in fact the conditional statement checks for 37 (5 + 32).

```
    if (rx != 5 + 32)
        THROW(0x6A87); // Wrong buffer size (has to be 32)
```

Re-test Notes

As of release 3.0.1, the `signer-certificate` component has been removed.

Unnamed Result Codes

Several functions throw result codes as raw hex values (e.g., see previous code that does `THROW(0x6A87);`). The use of raw hex values results in code that is difficult to read and validate. It is recommended to use named constants for such values to aid in readability. It is worth noting that the described value is in fact defined in *auth.h* as `#define AUTH_ERR_INVALID_DATA_SIZE (0x6A87)` .

Re-test Notes

As of release 3.0.1, the `signer-certificate` component has been removed, and non-named constant result codes appear to be uniformly replaced with more informative named result codes.

Uncommented enum/const Values

Building on the previous point, when constants are defined for various purposes, such as return codes, it would aid in readability to ensure that they are commented with a short description of their intent. For example, the `enum` values within *bc_err.h* are individually commented in a manner that makes their use clear. By comparison, the file *btctx.h* contains several definitions that might not be clear to someone without pre-existing knowledge of BTC transaction processing.

Re-test Notes

Aside from replacing raw hex return codes with named return codes, no additional changes were made in response to the above comment.

Ledger Development Best Practices

The Ledger developer portal provides a number of helpful advice and best practices. In particular, the following two resources are particularly relevant:

1. Security: https://developers.ledger.com/docs/nano-app/secure-app/
2. Common Pitfalls and Troubleshooting https://developers.ledger.com/docs/nano-app/troubleshooting/

Regarding the latter, Finding "Potentially Unsafe Exception Handling" details instances of exception handling within the codebase that violate requirements for exception handling. The other guidance items are relevant when an application crashes and must be diagnosed, but are not as prescriptive as the exception handling guidance.

The remainder of this subsection briefly surveys best practices from the Ledger Security Guidelines. Note that these guidelines are meant to guide apps wishing to be submitted to Ledger for listing on Ledger Live, but do provide meaningful general guidance for safely developing secure apps.

Perform Manual Code Reviews

A substantial portion of this report is the result of manual code review. Additionally, it is understood that code review is part of IOV Labs regular development process.

Automate Code Reviews with Static Code Analysis

A GitHub Workflow is defined to lint the library, see *lint-c.yml*. This runs the `lint-c` script in the repo, which primarily performs the following:

```
if [[ $1 == "exec" ]]; then
    if [[ "$(basename $0)" == "lint-c" ]]; then
        CLANG_ARGS="--dry-run --Werror"
    else
        CLANG_ARGS="-i"
    fi

    SRC_DIR="ledger/src"
    SEARCH_DIRS="$SRC_DIR/signer $SRC_DIR/ui $SRC_DIR/tcpsigner $SRC_DIR/common $SRC_DIR/
    ↳ signer-certificate"

    find $SEARCH_DIRS -name "*.[ch]" | \
    egrep -v "(bigdigits|bigdtypes|keccak256)\.[ch]$" | \
    egrep -v "ledger/src/ui/src/glyphs.[ch]" | \
    xargs clang-format-10 --style=file $CLANG_ARGS
```

A review of the script reveals that it primarily runs the `clang-format` tool on selected source files, but does not perform any meaningful static analysis on the code. A tool, such as the Clang Static Analyzer, as suggested in the Ledger docs, is recommended.

Avoid Warnings During Compilation

A cursory review of the build output reveals several warnings. A non-exhaustive list of different warning types encountered while building the UI follows:

- warning: incompatible integer to pointer conversion passing 'unsigned int' to parameter of type 'const void *' [-Wint-conversion]
- warning: incompatible pointer types passing 'unsigned char [20]' to parameter of type 'unsigned int *' [-Wincompatible-pointer-types]
- warning: invalid conversion specifier 'H' [-Wformat-invalid-specifier]
- warning: data argument not used by format string [-Wformat-extra-args]
- warning: result of comparison of constant 257 with expression of type 'volatile unsigned char' is always true [-Wtautological-constant-out-of-range-compare]
- warning: passing 'unsigned char [9]' to parameter of type 'const char *' converts between pointers to integer types with different sign [-Wpointer-sign]
- warning: incompatible integer to pointer conversion passing 'unsigned int' to parameter of type 'const void *' [-Wint-conversion]
- warning: unused variable 'i' [-Wunused-variable]
- warning: redefinition of typedef 'int8_t' is a C11 feature [-Wtypedef-redefinition]
- warning: comparison of integers of different signs: 'int' and 'unsigned int' [-Wsign-compare]
- warning: implicit declaration of function 'screen_display_init' is invalid in C99 [-Wimplicit-function-declaration]

These warnings do not necessarily indicate issues or vulnerabilities within the code, but in many cases may identify fragile or incorrect assumptions that can be remedied. Some of the identified warnings, such as the tautological comparison, may guard against future changes to the code. Similarly, signed/unsigned comparisons may work correctly in the current configuration, but trigger undefined behavior if data sizes change. Others, such as the use of `H` as a format string specifier may indicate actual bugs, where the intended behavior may be to use `X` to force hex output.

It is recommended to proactively address warnings in the current codebase, and to potentially fail builds on future warnings.

Re-test Notes

As part of release 3.0.1, commit ba41f89 addressed any remaining warnings within the codebase, and added the `-Werror` flag to the build process, ensuring that future warnings are also addressed. This addresses the recommendation above.

Test Requirements and Code Review

A GitHub Workflow is defined to run tests. The Ledger guide also recommends

1. Linting: It was noted earlier in this section that the linting process appears to simply run `clang-format`, and could be expanded.
2. Fuzzing: Some fuzzing functionality is provided in *ledger/fuzz*.
3. Static Analysis: As noted earlier, static analysis does not appear to be used.

Ask for External Security Audits

IOV Labs proactively engaged NCC Group for this purpose, and has been actively involved in the process.

Application flags

> Any use of a flag other than `APPLICATION_FLAG_BOLOS_SETTINGS` must be justified in the Makefile otherwise Ledger will not sign the application.

The UI uses `appFlags 0x248` which includes:

- `APPLICATION_FLAG_BOLOS_SETTINGS`: the application can read and modify system parameters such as the device's name.
- `APPLICATION_FLAG_GLOBAL_PIN`: the application can request a user PIN verification or query the number of tries left before the device erases its own memory.
- `APPLICATION_FLAG_BOLOS_UX`

These flags are necessary for the UI app to serve as the device UI, but it was noted that they are not explicitly documented within the `makefile`, as recommended.

Restrict Apps to Coin-Specific BIP32 Prefix

Confirming that arbitrary BIP32 paths cannot be used was a core goal of the assessment, and is reviewed in detail in Review Goals.

The recommendations also suggests that supported curves can also be restricted using the `--curve` parameter. It may be prudent to include `--curve secp256k1` in the `makefile` as an additional precaution, as this is the only curve currently supported.

Re-test notes

IOV Labs provided the following feedback:

> This would not improve powHSM's security, since we only allow our own specific apps to run on these devices, and have mechanisms in place to enforce this. This in turn implies that we know exactly what each of these applications do, and which curves it does and does not use.

The provided feedback is consistent with the implementation, and the only identified mechanism for a different curve to be used would be the authorization of a `Signer` update that does so, implying that this restriction is already enforced externally.

Never Store or Export Secrets Derived from Seed

Within the `Signer` and `UI` components, each call to `cx_ecdsa_init_private_key()` is followed by call to securely zero the private key data. The derived private key is then used either for signing, or to generate the corresponding public key. In each case, the private key is zeroed out immediately after use.

It was observed that within the `signer-certificate` component, keys do not appear to be zeroed out as above. Instead, global variables are defined to store the public and private keys, which are populated as part of the signing process based on the provided path; see *main.c*:

```
cx_ecfp_public_key_t publicKey;
cx_ecfp_private_key_t privateKey;
```

As part of the handler for `INS_SIGN`, these variables will be populated with the derived key via calls to `os_perso_derive_node_bip32()` and `cx_ecdsa_init_private_key()`. The derived private key data is not zeroed out here. Eventually, this handler leads to `io_seproxyhal_touch_approve()` where the actual sign operation takes place, also in *main.c*:

```
392 #else
393     tx = cx_ecdsa_sign((void *)&privateKey,
394                        CX_RND_RFC6979 | CX_LAST,
395                        CX_SHA256,
396                        result,
397                        sizeof(result),
398                        G_io_apdu_buffer);
399 #endif
400     G_io_apdu_buffer[0] &= 0xF0; // discard the parity information
401 
402     // Sign output buffer with attestation key
403     // attestation_len =
404     // os_endorsement_key2_derive_sign_data(G_io_apdu_buffer,tx,attestation);
405 
406     G_io_apdu_buffer[tx++] = 0x90;
407     G_io_apdu_buffer[tx++] = 0x00;
408     // Send back the response, do not restart the event loop
409     io_exchange(CHANNEL_APDU | IO_RETURN_AFTER_TX, tx);
410     // Display back the original UX
411     ui_idle();
412     return 0; // do not redraw the widget
413 }
```

As seen above, the private key persists in `privateKey`. Legacy code also appears to be present on lines 402-404. This was not documented as a formal finding as the impact was limited to the `signer-certificate` component, and therefore not part of the core scope.

Re-test Notes

As of release 3.0.1, the `signer-certificate` component has been removed, so the above comments are no longer relevant.

Avoid Blindly Signing Data

The current interface/implementation does not facilitate the signing of arbitrary data. To-be-signed hashes are derived from data within the library, and not supplied directly by the user.

Signing/Disclosing Keys Without user Approval

No interfaces or functionality are exposed for this purpose, and an explicit goal of this review involved the validation that the seed cannot be extracted from the device.

Don’t Roll Your Own Crypto Primitives

The library makes use of a SHA-256 implementation by Brad Conte, with light modifications to expose intermediate state; see *sha256.h*.

```
/*********************************************************************
 * Filename:   sha256.h
 * Author:     Brad Conte (brad AT bradconte.com)
 * Copyright:
 * Disclaimer: This code is presented "as is" without any guarantees.
 * Details:    Defines the API for the corresponding SHA1 implementation.
 *********************************************************************/
```

Note that the file comment references SHA1 instead of SHA2. A third-party Keccak (SHA3) implementation is leveraged in *keccak256.h*. The Ledger OS is leveraged for all other cryptographic functionality.

Avoid Exceptions for Cryptographic Code

> From the 2.0 version of the SDK (for Nano S, S Plus and X) every cryptographic function has a version that returns an error code instead of raising an exception. As an example cx_ecdsa_sign_no_throw performs the same computation as cx_ecdsa_sign but does not raise any exception and returns CX_OK if everything went fine.

> We recommend using all _no_throw equivalents when available, as the ones raising exceptions will be deprecated in a future SDK release.

The current library is not utilizing the "no throw" variants of these functions. While no specific issues were identified due to exception handling around cryptographic library calls, it is nevertheless recommended to plan a port to the newer functions before the old variants are deprecated.

Private Key Management

> You should minimize the code that works with private (ECDSA, RSA, etc.) or secret (HMAC, AES, etc.) keys. Importantly, you should always clear the memory after you use these keys. That includes key data and key objects.

The earlier section "Never Store or Export Secrets Derived from Seed" surveyed this requirement, and found the requirement to be met.

Corner Cases

The best practices guide concludes with a set of potential corner cases:

1. Be Wary of Untrusted Input,
2. Properly protect data you wish to cache on the host computer,
3. Do not allow the host to freely manipulate key-pairs.

These did not appear to be relevant, although the last item includes guidance around ensuring that state transitions for multi-message operations are strictly enforced. In general, this appears to be the case within the `Signer` and `UI` applications.

Unexpected UI State

While performing various tests on the Ledger hardware device, the NCC Group team came across a scenario leading to a possibly unexpected state. The steps to follow are presented below, where the device is assumed to have been freshly reset and the PIN and mnemonic have been set.

Setup

1. Recovery Mode is needed to install the signer and UI apps; in order to enter Recovery Mode, press right button when plugging in the Ledger.
2. A screen requiring inputting the PIN pops up.
3. Install the UI and signer app; both of which require the PIN to proceed with installation.
4. *Note:* restarting the Ledger normally results in the "User interface is not genuine" message.

5. *Note:* plugging in the Ledger with the right button pressed boots up in Recovery Mode. This appears to be in contradiction to the security goal that recovery mode cannot be accessed without wiping the device first. However, it should be noted that this is an edge case pertaining to the Ledger hardware itself and not specifically to the powHSM solution and that the resulting state the device is left in does not allow users to interact with it meaningfully.

Unexpected scenario

1. Now unplug the device and plug it in again in recovery mode.
2. Enter the wrong PIN 3 times. Device displays “Your device has been reset (3 wrong PIN)”.
3. But then, the ledger still opens the menu, where we can see the previously installed RSK Sign app (in addition to the Settings app).
4. The RSK app seems to work, it shows the “RSK: Waiting for msg” screen.
5. Now, restarting recovery mode does not require a PIN (which may not be unexpected since it’s consistent with the fact that we have wiped the PIN).
6. Navigating to the Settings app only shows two options (Device and Assistance).
7. However, when navigating through the menus Device, then Firmware, and going back to the Settings menu, the two other menu options are unlocked. Indeed, *now the Settings menu shows Display, Security, Device and Assistance*.
8. The Security menu is particularly unexpected, since it allows to change the PIN (and again, the PIN is wiped at this stage).
9. Now when trying to change the PIN, the Ledger prompts the user to enter the new PIN, but when it asks for confirmation of the old PIN, *the device hangs indefinitely*.
10. *Note*: this scenario was also confirmed on a fresh Ledger-only installation (without powHSM applications).

Re-test Notes

IOV Labs provided the following response regarding these observations:

> Even though the Signer application can indeed still be opened, it can no longer be operated through USB since a new mechanism was put in place to prevent such operation in case the device was wiped.