# Security Analysis of the `zcash-ledger` Hardware Wallet App

Taylor Hornby
zecsec@defuse.ca
ZecSec

September 20, 2023
Report Version 1
Audit performed in May/June 2023

# Contents

# 1 Introduction

This report documents the results of a 15-day security audit of the `zcash-ledger` Ledger wallet app. `zcash-ledger` enables support for Zcash transactions on Ledger hardware wallets [3]. The app is developed by Hanh Huynh Huu with funding provided by the Zcash Community Grants program [4].

The `zcash-ledger` app supports generating Zcash transctions with both transparent and shielded inputs and outputs. It supports the Sapling and Orchard shielded pools on NanoS+ and NanoX devices, only Sapling on NanoS devices, and Nano Stax devices are not supported.

We found one *Medium*-severity issue that could put funds at risk: side-channel defenses in the signing algorithm may be inadequate and could leak information about spending keys to a malicious host.

We found several other *Medium*-severity issues: (a) because the Ledger device's operating system does not support ZIP 32, the wallet is not cross-compatible with other Zcash shielded wallets, (b) a malicious host can "burn" some of the user's funds by producing invalid ciphertexts, and (c) a secure-UX issue where unified addresses are not displayed properly when the user is confirming a transaction on the hardware device.

We also report several *Low*-severity issues, which are mainly minor code improvement suggestions, and pose little risk to users.

In the next section we describe the audit's scope. In Section 2, we lay out a threat model for shielded hardware wallets. In Section 3, we describe the vulnerabilities we found. In Section 4 we make recommendations to improve `zcash-ledger`'s security. Section 5 lists some good things we found during the audit. Section 6 lists our suggestions for focus areas of future audits, and Section 7 concludes.

This audit began while `zcash-ledger` was still in early development. As a result, we found several security bugs in pre-release versions of the code. These were all fixed before the first formal release, so we report them separately in Appendix A.

## 1.1 Scope

Our review focused on the code in the `hhanh00/zcash-ledger` GitHub repository. We audited several versions of this codebase, concluding with a final pass over release v1.0.1, which has the commit hash:

`2acd2a66010fa80c1e906367fc7a953e7475d794`

Our focus was to find any bug which either (a) would put the user's spending keys at risk of theft or (b) would make it possible to sign a transaction with output components or fees that were not approved by the user. This included looking for remote-code-execution style attacks against the hardware wallet app, improper implementations of cryptography, and bugs in the transaction confirmation process.

This audit *did not* consider:

- Attacks by a malicious host computer on the user's privacy.

- The security of Ledger devices themselves.
- The security of the BOLOS operating system that runs on Ledger devices.
- The security of the cryptographic (and other) libraries in the Ledger SDK.
- Security/privacy issues in any host wallet that interacts with the hardware wallet app.
- Code that is only compiled into the app when various `TEST` or `DEBUG` flags are set.

While we reviewed all of the code in `hhanh00/zcash-ledger`, time constraints prevented us from adequately reviewing the elliptic curve implementations that are used for Sapling and Orchard signing. We recommend further review of the curve implementations as future work in Section 6.1.

# 2 Threat Model

To define a simplified threat model for a Zcash hardware wallet, we consider the following scenario:

1. The user initially sets up their hardware wallet using a host which is fully compromised by the attacker. The user is careful to back up their seed phrase and never enter it into the compromised host.
2. The user sends funds to their hardware wallet, using the addresses displayed on its dedicated screen.
3. The user spends funds from the wallet, carefully checking the transaction details displayed on the device's screen are correct, and only approving the transaction on the device when all details are as expected.

Within this scenario, we define the threat model in terms of what expectations the user has regarding the attacker who has compromised their host, following the process of Invariant-Centric Threat Modelling [1].

We expect that the attacker *CAN*:

- Prevent the user from using their wallet on that compromised host (but not others).
- Steal viewing keys which let the attacker permanently compromise the privacy of the wallet's addresses.
- Cause funds being spent by the user to be stolen, if the user did not carefully verify the transaction details on the device's screen.
- Cause funds that the user is receiving to be stolen, if they did not verify their address on the device's screen or if they used the compromised host to transmit their address (e.g. the attacker could replace the user's address with their own after they paste it into an exchange's website).
- Cause transactions to be generated in a way that weakens privacy (e.g. by causing randomized signatures to not be randomized, or by tagging transactions with additional identifying data).

The user expects that the attacker *CANNOT*:

- Obtain the spend authority keys for any of the user's addresses.
- Cause the user's funds to become unspendable.
- Spend the user's funds without physical approval on the hardware device.
- Produce a transaction whose semantics are different in any way from what was approved through the device's display, including changing the destination of funds being spent, the amount of funds being spent, or the fee amount.
- Cause the hardware wallet to display, as its own address, any other address, such as one an attacker controls.
- Cause an official-looking message to be displayed on the hardware wallet's display (which could be used for phishing).
- Permanently prevent the hardware wallet from functioning on another uncompromised host.

Beyond these specialized considerations for hardware wallets, a host wallet that interfaces with the Ledger wallet app is expected to fall under the scope of the Zcash Wallet App Threat Model [2]. (Note that we did *not* review any host wallet as part of this audit.)

### 2.1 A caveat about transaction confirmation

The `zcash-ledger` app does not ask the user to confirm transaction *inputs*. The host is completely free to specify whichever inputs it would like, and these are *unverified* by the hardware wallet. This means that a malicious host could spend different notes than the user intends, or even spend notes/UTXOs owned by someone else as part of the user's transaction, and this will not be visible to the user in the on-device confirmation process.

Balance security therefore rests on the idea that the user confirms each transaction output's destination and amount, as well as the fee amount, so that an upper bound on the amount of the user's funds being spent *is* confirmed through the hardware device. Note that this design relies on the Zcash consensus rules to reject transactions which have calculated their fees or pool balances incorrectly.

## 3 Security & Privacy Findings

This section describes the security and privacy issues that were found during the review.

Each issue has been given a priority rating, determined informally by combining the severity of the issue's impact on users with its likelihood of being exploited in practice.

A "*Critical*" issue is a vulnerability that can definitely be exploited to impact many users with devastating consequences. "*High*" means a vulnerability that is likely to have a severe impact on many users. "*Medium*" means a vulnerability of moderate impact or one that may only be exploitable in special circumstances. "*Low*" means a vulnerability whose exploitation would have very little impact on any user or which is unlikely to ever be exploited in practice.

*Critical* and *High*-severity issues must be fixed as soon as possible to protect users. *Medium*-severity issues are sometimes safe to defer, and *Low*-severity issues are almost always safe to defer.

## Issue A Potential side-channel leakage of signing keys

**Severity:** *Medium*

In the process of producing transaction signatures, a double-and-add algorithm is used to perform the elliptic curve scalar multiplication.

Unless double-and-add is implemented carefully, side-channels (timing, power usage, etc.) can leak the signing key by making the "add" case (signing key bit is 1) distinguishable from the "don't add" case (signing key bit is 0).

The two implementations of double-and-add in `zcash-ledger`, for Sapling and Orchard respectively, attempt to prevent side-channel leakage by always adding, adding the identity point in the "don't add" case:

```
void en_mul(jj_e_t *pk, jj_en_t *G, cx_bn_t sk) {
    bool bit;
    e_set0(pk);
    jj_en_t id; alloc_en(&id);
    e_to_en(&id, pk);
    // Skip the higest 4 bits as they are always 0 for Fr
    for (uint16_t i = 4; i < 256; i++) {
        cx_bn_tst_bit(sk, 255 - i, &bit);
        e_double(pk);
        PRINTF("*");
        if (bit) {
            PRINTF("+");
            een_add_assign(pk, G);
        }
        else
            een_add_assign(pk, &id); // make it constant time
    }
    destroy_en(&id);
    PRINTF("\n");
    // ...
}
```

```
void pallas_base_mult(jac_p_t *res, const jac_p_t *base, fv_t *x) {
// ...

int j0 = 1; // skip highest bit
for (int i = 0; i < 32; i++) {
      uint8_t c = (*x)[i];
      for (int j = j0; j < 8; j++) {
            // print_bn("acc x", acc.x);
            pallas_double_jac(&acc);
            if (((c >> (7-j)) & 1) != 0) {
                  pallas_add_jac(&acc, &acc, &b);
                  // print_bn("acc x", acc.x);
                  // print_bn("acc y", acc.y);
                  // print_bn("acc z", acc.z);
            }
            else
                  pallas_add_jac(&acc, &acc, &id);
      }
      j0 = 0;
}

// ...
}
```

Each of these implementations has an "else" case which attempts to make the double-and-add algorithm constant-time by adding the identity point.

However, this is not adequate to make the implementations constant-time: the code is still branching on the bit of the signing key, and in the two different code paths, different memory is accessed. The branching and different memory accesses may be enough to leak information about the signing key's bits through timing analysis, depending on the architecture of the Ledger device.

In addition, `pallas_add_jac()` special-cases the identity point in order to skip running the addition algorithm when the identity point is being added. So in the case of Orchard, there are definitely observable timing differences:

```
1  void pallas_add_jac(jac_p_bn_t *res, const jac_p_bn_t *a, const jac_p_bn_t *b) {
2      if (pallas_is_identity(a)) pallas_copy_jac_bn(res, b);
3      else if (pallas_is_identity(b)) pallas_copy_jac_bn(res, a);
4      else {
5      // ...
```

To exploit this, a malicious host would need to produce many signatures and measure the time it takes to create them. The signature-producing APIs that the app exposes do not limit the amount of times a signature can be generated, so this may be feasible.

The signing keys are randomized (by adding a random $\alpha$ that's known to the host), which makes the attack easier: without adding $\alpha$, the timing would be roughly constant for each signature and not much information would be leaked in total. With a random $\alpha$ added each time, the running time leaks information about how many bits are 1 *after adding* $\alpha$, so collecting many samples with different $\alpha$'s leaks more information about the non-randomized signing key.

This should be fixed by doing a constant-time *selection* between the accumulator and the identity point that involves no branching at all on bits of the signing key. MIT-licensed code for doing a constant-time conditional copy (which can be used to implement a conditional selection) can be found in the `CTTK` library:

https://github.com/pornin/CTTK/blob/master/src/oram1.c#L29

## Issue B Keys are not derived according to ZIP 32

**Severity:** *Medium*

The operating system on Ledger devices only supports standard BIP 32 derivation using the `os_perso_derive_node_bip32()` API. This API does not support a way to derive keys and addresses compliant with ZIP 32.

ZIP 32 key derivation is similar in structure to BIP 32 key derivation, but ZIP 32 involves hashing the root seed with BLAKE2b with different personalizations for Sapling and Orchard. This is not possible to accomplish using the `os_perso_derive_node_bip32()` API, and the root seed is inaccessible to the Ledger app, since allowing access to the root seed would grant the app the ability to generate the spending keys for other coin types. The only way to properly support ZIP 32 on Ledger would be for Ledger to upgrade their OS to support ZIP 32 key derivation.

`zcash-ledger` works around this problem by generating the transparent spending key according to BIP 32 using the above-mentioned API. Then, the transparent spending key is hashed with

BLAKE2b using personalizations of `ZSaplingSeedHash` and `ZOrchardSeedHash` to get the Sapling and Orchard spending keys, respectively:

```
static int derive_spending_key(uint8_t *spk, uint8_t account) {
    derive_tsk(spk, account);

    cx_blake2b_init2_no_throw(&G_context.hasher, 256,
                              NULL, 0,
                              (uint8_t *) "ZSaplingSeedHash", 16);
    cx_hash((cx_hash_t *) &G_context.hasher,
            CX_LAST,
            spk, 32,
            spk, 32);
    return 0;
}
```

```
void orchard_derive_spending_key(int8_t account) {
      ui_display_processing("o-key");
      derive_tsk(spending_key, account);

      cx_blake2b_init2_no_throw(&G_context.hasher, 256,
                                NULL, 0,
                                (uint8_t *) "ZOrchardSeedHash", 16);
      cx_hash((cx_hash_t *) &G_context.hasher,
                  CX_LAST,
                  spending_key, 32,
                  spending_key, 32);
      // ...
```

This creates two problems.

Firstly, `zcash-ledger` will not be compatible with any other shielded Zcash wallet. If a user attempted to recover their `zcash-ledger` funds using a different wallet, it would appear that they have zero balance. Support for `zcash-ledger`'s key derivation scheme could be added to other wallets, but it would double the scanning effort needed to find the user's transactions (since the wallet doesn't know whether the seed phrase was used in a ZIP 32 wallet or in `zcash-ledger`). Importing a seed phrase generated on a ZIP-32-compliant wallet into a Ledger device would also not work.

Secondly, deriving the Sapling and Orchard keys from the transparent key puts them at greater risk. If there were a bug in just the transparent transaction signing code which leaked the transparent key, the Sapling and Orchard keys could be derived from it, and then even the user's shielded funds could be stolen.

To prevent the first problem, and avoid mass confusion created by wallets doing key derivation differently, the release of `zcash-ledger` should be blocked on either (a) Ledger adding support for ZIP 32, or (b) ZIP 32 being amended to work with Ledger devices in a safe way.

To fix the second problem, the Sapling and Orchard keys need to be domain-separated from the transparent key, so that none of the three keys can be derived from each other. A hack to do this would be to use a different coin type or range of account numbers when generating the Sapling and Orchard keys, but this could lead to even *more* confusion and incompatibility between wallets. This should be fixed properly with ZIP 32 support from Ledger, or by an amendment to ZIP 32 with a standardized way to do this safely.

## Issue C Burning funds by producing invalid output ciphertexts

**Severity:** *Medium*

Sapling outputs and Orchard actions each contain an output ciphertext. These ciphertexts are encrypted to transaction recipients' public keys, and the information contained in them is necessary for the recipients to subsequently spend the funds they receive.

The `zcash-ledger` app trusts the host to provide these ciphertexts and their associated ephemeral public keys, and does not verify them. If a malicious host provides incorrect ciphertexts or ephemeral public keys, the output notes will be undetectable and unspendable by the recipient. A malicious host can take advantage of this to effectively burn a `zcash-ledger` user's funds.

`zcash-ledger` does not ask the user for confirmation of Sapling or Orchard outputs that are destined to addresses owned by the hardware wallet itself. A malicious host can take advantage of this to burn *all* of a user's shielded funds by the following attack:

1. Wait for the user to start creating a legitimate transaction.
2. As the transaction is being created, add Sapling outputs and Orchard actions which spend all remaining funds to the wallet's own address.
3. Provide the `zcash-ledger` app with random data instead of the real output ciphertexts or ephemeral public keys, and then delete all of the data that should have been contained in those ciphertexts from memory.
4. Construct the remainder of the transaction as normal, and wait for the user to approve it for signing.
5. Obtain the transaction signatures and broadcast the transaction to the Zcash network.

The additional Sapling outputs and Orchard actions would not be made visible to the user, since they are being sent to the wallet's own address. Even though the ciphertexts and ephemeral public keys have been replaced with random ones, the transaction will still be valid according to Zcash's consensus rules. All of the user's notes will have been spent, and there will be no way for the user to recover the funds, because the information required to spend the new notes has been deleted.

To fix this, `zcash-ledger` needs to verify that the ephemeral public key is derived properly and that the note ciphertext has been computed properly.

Verifying just the first 52 bytes of the ciphertext is sufficient to ensure that funds would always be *recoverable*, but the entire ciphertext needs to be verified to ensure that the funds will actually be *detected and received* by the destination wallet. This is because if only the first 52 bytes are checked, a malicious host could flip some other bit (e.g. in the part of the ciphertext corresponding to the memo), which would cause the ciphertext integrity check to fail, leading most wallets to ignore the transaction.

This implies that `zcash-ledger` needs to be responsible for computing trusted values for `sapling_outputs_memos_digest` and `orchard_actions_noncompact_digest` (as defined in ZIP 244) as well; in the current implementation, the host is trusted to provide them.

## Issue D Addresses cannot be confirmed when sending to a UA

**Severity:** *Medium*

When asking the user to confirm an output to a transparent address, `zcash-ledger` shows the `t...` address on its screen. Similarly, for a Sapling output, a `zs...` address is shown, and for Orchard, a unified address (UA) with one Orchard receiver is shown.

This creates a secure usability problem when a user is attempting to send to a unified address. When the user sends funds to a unified address, the confirmations shown on the device will be the individual transparent, Sapling, or single-receiver Orchard addresses for the individual receivers contained in the destination unified address, rather than the destination unified address itself.

Without decoding and re-encoding the destination unified address themselves, the user has no way to ensure that the addresses shown on screen match their intended destination address. Users working around this issue may become accustomed to *not* verifying the addresses in the confirmation process, and a malicious host could take advantage of this behavior to steal funds.

To fix this, the APIs for adding outputs to a `zcash-ledger` transaction need to be expanded to support unified addresses. When a destination address is a unified address, `zcash-ledger` needs to accept all of its receiver components and re-encode the unified address for display, ensuring that the transparent, Sapling, or Orchard address that the current output is being sent to is contained in the unified address that gets displayed.

## Issue E Stack canary checking is frequently disabled

**Severity:** *Low*

Ledger devices only have a small amount of stack space, so Ledger documentation recommends frequently checking a stack canary to detect stack overflows. Without these checks, if too much stack space is used, it will start to overwrite other older data on the stack.

In many of `zcash-ledger`'s build configurations, stack canary checking is disabled. By testing releases and not seeing buggy behavior, we can have some confidence that the stack is not overflowing. However, updates to Ledger's SDK or OS may cause the app to use more stack space. If this happens, a stack overflow could occur in production and go undetected.

The most likely result would be a crash that poses little risk to users, but there is a chance that a stack overflow could be exploited by a malicious host to execute a remote-code-execution attack against the device and steal keys.

We recommend checking for stack overflows, at least around calls to functions defined outside of `zcash-ledger`, which use potentially-changing amounts of stack space.

## Issue F Missing error handling

**Severity:** *Low*

Calls to Ledger APIs should be evaluated for proper error handling. In particular:

- `cx_get_random_bytes()`, used for generating signature nonces and CSPRNG seeds, can theoretically fail and needs error handling. The alternative `cx_rng_no_throw` can be used instead.

- Math functions like `cx_bn_mod_add` can fail, e.g. due to out-of-memory situations, and therefore need error handling.
- The AES calls in `ff1.c` can also fail.

While it's unlikely for these calls to fail, a missed failure could lead to the use of a weak nonce or lead to an incorrect calculation. To fix this, ensure that proper error handling is added to all calls that may fail.

## Issue G Minor code issues and suggestions

**Severity:** *Low*

### Issue G.1 Wipe sensitive data in `G_store` after use

`G_store` is a global variable holding a `temp_t` union which is repurposed as storage space for various tasks. In particular, during transparent signing it holds the transparent signing key and signature nonce:

```
typedef struct {
      // ...
    union {
      // ...
        struct { // transparent sign
            uint8_t sig_hash[32];
            uint8_t tsk[32];
            uint8_t signature[64];
            uint8_t rnd[32];
        };
      // ...
    };
} temp_t;
```

After the transparent signature has been generated, the key and nonce are left in `G_store`. We carefully verified that no other uses of the union could leak these secret values. We recommend zeroing them immediately after use to make it obvious they won't be leaked through other uses of the union.

### Issue G.2 `SET_S_NET` and `SET_O_NET` are not actually optional

A code comment says `SET_S_NET` and `SET_O_NET` are optional calls:

```
// These functions are optional but must be called before confirm_fee
int set_s_net(int64_t balance);
int set_o_net(int64_t balance);
```

However, they are not reset to zero in either `reset_app()` nor in the handler for `INIT_TX`. So, when a new transaction is created without calling `SET_S_NET` and `SET_O_NET`, the Sapling and Orchard net values from the previous transaction will be used.

### Issue G.3 `reset_app()` and `INIT_TX` do not zero `G_context`

`zcash-ledger` stores information needed during the construction of a transaction in a global `G_context` variable.

When `zcash-ledger` encounters an unexpected condition (which could be a sign of an attempted attack by a malicious host), a common way to handle it is to call `reset_app()`. `reset_app()` only sets the stage back to `IDLE`, however, and does not zero any of the context. This means that much of the context from the previous transaction will still be in `G_context` while the next transaction is being created.

We did not find any security bugs as a result of this, but the code would be simpler to analyze if `reset_app()` zeroed as much of the context as possible, to ensure that no old data is leftover during the creation of new transactions.

The same applies to `INIT_TX`: much of `G_context` could be zeroed and/or re-initialized in the `INIT_TX` handler as well.

### Issue G.4 Undefined behavior when calculating `t_net`

In the calculation of `t_net`, a `uint64_t` gets cast to an `int64_t`:

```
1  int add_t_input_amount(uint64_t amount) {
2        // ...
3        G_context.signing_ctx.has_t_in = true;
4        G_context.signing_ctx.t_net += (int64_t)amount;
5        // ...
```

This is technically undefined behavior. To fix this, check that `amount` is within the range of positive values an `int64_t` can hold before performing the cast.

Our recommendation to add compiler warning flags (Recommendation 4.4) would catch similar issues.

### Issue G.5 `ROTATE` and `PLUS` macros missing parentheses

In `chacha.c`, the `ROTATE` and `PLUS` macros are missing parentheses around the use of their arguments in their definitions:

```
1  #define ROTATE(v, n) (cx_rotl(v, n))
2  #define XOR(v, w)    ((v) ^ (w))
3  #define PLUS(v, w)   ((uint32_t) (v + w))
```

If these macros are used on complex expressions, order-of-operations could cause them to produce incorrect results. In the implementation of ChaCha, they are never used on complex expressions, so there is no actual bug, but it's worth fixing the macros anyway.

### Issue G.6 `GET_PROOFGEN_KEY` does not ensure `derive_default_keys()` has been called

A host can call `GET_PROOFGEN_KEY` before any of the other APDU handlers that call `derive_default_keys()`. When this happens, invalid proof generating keys will be returned (possibly zeroed or uninitialized memory). Fix this by ensuring `derive_default_keys()` has been called.

See also Recommendation 4.5 where we suggest being stricter about enforcing the order of APDU calls.

#### Issue G.7 Some APDU handlers are missing the no-data check

Many APDU handlers contain the following code to reject calls with data when none is expected:

```
1  if (cmd->lc != 0)
2        return io_send_sw(SW_WRONG_DATA_LENGTH);
```

However, this check is missing for some APDU handlers, including `GET_PROOFGEN_KEY`, `CONFIRM_FEE`, `CHANGE_STAGE`, and others. There is no bug as a result of this, but it may be worth adding the checks.

# 4 Recommendations

## 4.1 Better test-vector testing of cryptographic components

Reviewing implementations of cryptographic algorithms like the elliptic curves, FF1, Pedersen and Sinsemilla hashes, and so forth, is time-consuming and error prone.

A much greater confidence in the correctness of these algorithms can be gained by implementing tests which compare the input/output of the implementations in `zcash-ledger` against a wide set of test vectors produced by a different implementation such as in `librustzcash`. Randomized testing, on top of test vector cases designed to exercise edge cases, can increase confidence further.

Some good testing already exists in `zcash-ledger`, but it could be improved. For example, bugs in the FF1 implementation would not lead to invalid transactions, but could make funds hard to recover. We therefore recommend strengthening the automated testing of all cryptographic algorithms.

## 4.2 Reduce the likelihood of accidental use of `TEST`/`DEBUG` builds

If the `TEST` or `DEBUG` compile flags are set and `zcash-ledger` is built, the resulting app is totally insecure. This is by design to allow for testing: all on-device confirmations can be bypassed, and secret keys can be replaced with fixed constants.

Some defenses should be added to the code to absolutely eliminate the possibility of anyone accidentally side-loading a test build of `zcash-ledger` onto their device. This could take the form of a warning message that frequently gets displayed, e.g. prefix all of the normal text with a "WARNING: INSECURE TEST BUILD" string.

## 4.3 Compatibility testing with other wallets

After Issue Issue B is fixed, compatibility testing with other Zcash shielded and transparent wallets should be implemented. This will prevent accidental loss of funds in case any future bugs make `zcash-ledger` incompatible with those other wallets.

## 4.4 Increase compiler warning level

By default `zcash-ledger`'s build system does not pass any additional `-W...` warning flags to the compiler. By adding `-Wall` or a smaller custom set of warning flags, the compiler can be told to check for various kinds of (likely) bugs. Once these warnings are fixed, we recommend adding `-Werror` as well.

## 4.5 Enforce stage-specific operations more strictly

To construct a transaction using `zcash-ledger`, the process proceeds through a series of stages. The host first adds transparent inputs and outputs, then Sapling outputs, then Orchard actions, then confirms the fee, and so on. At each change of stage, the host must call the `CHANGE_STAGE` APDU to advance to the next stage.

Most of the crucial APDU commands, such as for adding transaction outputs, are strictly enforced to occur during they stage in which they are expected. However, several APDU commands, which could be stage specific, can be executed by the host at any time. The state machine in `zcash-ledger` should be hardened to disallow this behavior, i.e. all but the most general APDU commands should be gated to a specific stage.

Doing this will place more requirement on the host to make calls in a specified order, but will simplify the security analysis of the code.

Areas for hardening that we identified are:

1. While in stage `FEE`, the host can call the `CONFIRM_FEE` APDU to have the device calculate the SIGHASH and show the fee confirmation. The stage will remain in `FEE` until the user denies (stage returns to `IDLE`) or approves (stage advances to `SIGN`). While the user is confirming the fee, the stage is still in `FEE`, so the host can modify parts of the transaction (e.g. calling `SET_S_NET`) then call `CONFIRM_FEE` again to show another fee confirmation. This behavior is unexpected, and should be ruled out.
2. APDUs like `SET_S_NET`, `SET_O_NET`, `SET_*_MERKLE_PROOF`, and more can be called in any stage. While allowing extra flexibility makes developing the host wallet easier, strictly enforcing a stage for these APDUs would make it easier to analyze `zcash-ledger`'s code.

In the case of (1), we were concerned that a race condition could potentially be used by a malicious host to create an exorbitantly-large fee. The idea was to (1) lie about the Sapling net value and call `CONFIRM_FEE` so that the fee confirmation looks normal but the transaction would be invalid, (2) while the user is confirming the fee, call `SET_S_NET` again to provide the correct Sapling net value, (3) call `CONFIRM_FEE` again to recalculate the SIGHASH. The hope was that a race condition could set the stage to `SIGN` from the first fee confirmation *after* the SIGHASH had been recalculated with the exorbitant fee. This turns out not to work, but the reason why depends on internal and possibly-changing details of how Ledger apps handle APDUs. We recommend changing the code to make it very obvious this kind of attack could never work. One way to do this is to have `CONFIRM_FEE` set the stage to a new `FEE_CONFIRMING` stage immediately, so that `CONFIRM_FEE` cannot be called twice.

# 5 Good Things

## 5.1 Fast security response

In the process of `zcash-ledger`'s development, we reported several issues to the author, all of which were fixed rapidly, often the same day. Some of the issues were even noticed and fixed before we reported them.

## 5.2 Integration testing

Using the Ledger device simulator, `zcash-ledger` has integration tests for most of its features. This is great for ensuring the app works as expected and some of the tests would even catch certain kinds of cryptographic implementation bugs.

# 6 Future work

## 6.1 Closer review of elliptic curve implementations

In this review, we did not thoroughly check the elliptic curve implementations. The correctness of these implementations is crucial for security, since they are used in the transaction signing process. Future security audits should review them closely.

## 6.2 More analysis of `zcash-ledger`'s state machine

In the process of interacting with the `zcash-ledger` app, the host moves the app through a series of stages. The `zcash-ledger` app informally implements a state machine that moves through these stages.

State machine bugs are a common source of error in cryptographic software, and `zcash-ledger`'s state machine's correctness is crucial to the app's transaction confirmation process. We found two *almost-bugs* in the state machine (the issue in Appendix A.3 and the almost-bug described in Recommendation 4.5). Therefore, we recommend further review of `zcash-ledger`'s state machine.

# 7 Conclusion

Let's summarize the issues again:

We found one *Medium*-severity issue that could put funds at risk: side-channel defenses in the signing algorithm may be inadequate and could leak information about spending keys to a malicious host.

We found several other *Medium*-severity issues: (a) because the Ledger device's operating system does not support ZIP 32, the wallet is not cross-compatible with other Zcash shielded wallets, (b) a malicious host can "burn" some of the user's funds by producing invalid ciphertexts, and (c) a secure-UX issue where unified addresses are not displayed properly when the user is confirming a transaction on the hardware device.

We also reported several *Low*-severity issues, which are mainly minor code improvement suggestions, and pose little risk to users.

Many of the issues have simple fixes, but the incompatibility with ZIP 32 should be a concern to the Zcash community. We either rely on Ledger to implement proper ZIP 32 support in their operating system, or we need to complicate ZIP 32 itself with a workaround to support Ledger hardware wallets safely. If neither of these paths succeed, we will be left with a confusing mess where wallets derive keys from seeds differently, which will create a terrible user experience.

This audit *did not* adequately review the elliptic curve implementations in `zcash-ledger`. This has been left for future work.

## Acknowledgements

This security analysis was performed as part of the Zcash Ecosystem Security Grant [5, 6] funded by Zcash Community Grants [4]. Thanks to the Zcash Grants Committee and the broader Zcash community for supporting the security of privacy-enhancing open-source software.

Thanks to Daira Hopwood and Jack Grigg for guidance on how to remediate the ZIP 32 incompatibility issue.

## References

[1] Invariant-Centric Threat Modeling.
https://github.com/defuse/ictm.

[2] Zcash Wallet App Threat Model.
https://zcash.readthedocs.io/en/latest/rtd_pages/wallet_threat_model.html.

[3] zcash-ledger.
https://github.com/hhanh00/zcash-ledger.

[4] Zcash Community Grants.
https://zcashcommunitygrants.org.

[5] Zcash Ecosystem Security Grant.
https://forum.zcashcommunity.com/t/zcash-ecosystem-security-lead/42090, .

[6] ZecSec: Zcash Ecosystem Security.
https://zecsec.com/, .

# A Bugs Discovered During Early Development

In this section, we report some bugs that were found in pre-release versions of the code. All of these issues were resolved prior to the first release (v1.0.1).

## A.1 Predictable nonces used while testing ECDSA signatures

**Severity:** *Critical*

In an in-development version of the code, a constant of 32-bytes of repeated 3s was used as the seed for the random number generator to generate the ECDSA nonces for the shielded pool signatures.

This would have let the host predict the nonces, and so a malicious host could have recovered the secret spending keys from the transaction's signatures.

This was fixed by generating the nonces at random using the device's secure random number generator (except for test builds, which use predictable nonces to enable testing).

## A.2 Fee confirmations could be bypassed by the host

**Severity:** *High*

In an in-development version of the code, the testing feature which allows bypassing confirmations was not disabled in production builds for the fee confirmation.

This would have let a malicious host send arbitrary amounts of the user's funds into the transaction fee. In cooperation with a miner, this would have allowed for theft of funds.

It has been fixed by only allowing the confirmation bypassing to work in test builds.

## A.3 Steal-by-fee by improperly advancing the `stage` state

**Severity:** *High*

The host advances `zcash-ledger`'s state machine by calling the `CHANGE_STAGE` APDU, which calls the `change_stage()` function:

```
1  int change_stage(uint8_t new_stage) {
2      if (new_stage != G_context.signing_ctx.stage + 1) {
3          reset_app();
4          return io_send_sw(SW_BAD_STATE);
5      }
6
7      cx_hash_t *ph = (cx_hash_t *)&G_context.hasher;
8
9      switch (new_stage) {
10         case T_OUT:
11         // ...
12         // ... cases that handle stages S_OUT, O_ACTION, and FEE
13         // originally there was no default case
14     }
15     G_context.signing_ctx.stage = new_stage;
16
17     return io_send_sw(SW_OK);
18 }
```

The `new_stage` parameter comes directly from the host.

Originally, the switch statement did not have a `default` case to detect bad state transitions. Therefore, a malicious host could:

1. Wait for the user to initiate a legitimate transaction.
2. Add extra inputs to the transaction so that it would have an enormous fee.
3. Wait for the user to confirm all of the transaction outputs on the device. Now `stage` will be `FEE`.
4. Initiate the fee confirmation, which computes the transaction's SIGHASH for signing and shows the fee to the user, but `stage` should not change to `SIGN` unless the user approves.
5. Call `CHANGE_STAGE` with `new_stage = SIGN = FEE + 1` before the user rejects the bad fee. Now `stage` is `SIGN` and signatures for the transaction can be extracted by calling the signing APDUs.

In other words, a malicious host could have sent arbitrary amounts of the user's funds into a transaction fee if the user was not quick enough to disapprove of the displayed bad fee. In cooperation with a miner, this could have been used to steal funds.

This was fixed by adding a `default` case that rejects bad state transitions:

```
      // ...
        default: // should have been caught by the check at the top
            return io_send_sw(SW_INVALID_PARAM);
      // ...
```

## A.4 Out-of-bounds memory access in the FF1 implementation

**Severity:** *Low*

In an old version of the FF1 implementation, a `memmove` made an out-of-bounds write:

```
int ff1(uint8_t *d, const uint8_t *dk, uint8_t *di) {
      // ...
      memset(P, 0, 16);
      P[9] = i;
      memmove(&P[10], b, 6);
      // block depends on the round number and b
      error = cx_aes_iv_no_throw(&aes_key, CX_ENCRYPT | CX_PAD_NONE | CX_CHAIN_CBC | CX_LAST, (↩
          uint8_t *)R, 16,
          (uint8_t *)P, 16, (uint8_t *)R, &out_len);
      if (error) return error;
      memmove(d, R, 16);
      // ...
```

The `d` written to in the last line was allocated with a size of only 11 bytes in `orchard.c`, so the 16-byte write would overflow:

```
// ...
uint8_t d[11];
memset(d, 0, 11);
```

```
4   ff1(d, G_context.orchard_key_info.dk, d);
5   // ...
```

This code can be found in commit `afc3a2b0b0007639a865b9415470591abbecd17b`. It has been fixed by entirely removing the `ff1()` function, replaced by `ff1_inplace()` which does not appear to suffer from the same issue.