# Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

MANTA NETWORK

## Manta Chain

Veridise Inc.

April 28, 2023

► **Prepared For:**

Manta Network
https://manta.network/

► **Prepared By:**

Shankara Pailoor
Jon Stephens
Burak Kadron
Jacob Van Geffen
Kostas Ferles
Daniel Dominguez
Benjamin Sepanski

► **Contact Us:** contact@veridise.com

► **Version History:**

April 27, 2023 Initial Draft

# Contents



 

# 1 Executive Summary

From Mar. 6, 2023 to April. 17, 2023, Manta Network engaged Veridise to review the security of their blockchain implementation, henceforth referred to as Manta Chain. Manta Chain is a Substrate-based Polkadot parachain that exposes a protocol called MantaPay, whereby clients (such as other parachains or end-users) can trade and deposit assets privately. Veridise conducted the assessment over 30 person-weeks, with 5 engineers reviewing code over 6 weeks on commit `45ba60e1d`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers, namely static analysis and fuzz testing, as well as extensive manual auditing.

**Code assessment.** Since Manta Chain is developed fully open-source, Manta Network developers provided Veridise auditors the link to Manta Chain's public github repository along with the commit to be reviewed. In addition to the code, Veridise auditors were given several detailed, easy-to-read documents outlining the intended behavior of Manta Chain including a whitepaper, a formal specification of the MantaPay protocol, along with links to additional documentation hosted on their website. The Manta Chain code is also well documented and contains detailed, yet clear, comments specifying the intended behavior of the corresponding code. The code is also organized nicely with different components separated into independent Rust crates; as an example, each pallet is separated into its own crate.

Every crate in the Manta Chain codebase also had an accompanying test suite which exercised all the functions and security-critical paths in the package. Veridise auditors found the test suites to be very helpful as they (1) illustrated the expected ways of invoking the external calls exposed by Manta Chain and (2) helped when developing a fuzzer for Manta Chain.

There are two concerns worth noting about the codebase. First, parts of Manta Chain were copied from other Polkadot parachains. In particular, the staking logic was derived from Moonbeam and the tx-pause pallet was taken from Acala. Both of these blockchains have had audits and the borrowed code is of high quality; however, the developers and users of Manta Chain should keep in mind that if bugs are discovered in the original implementations then they could easily be present in Manta Chain's version and the fixes should be propagated. Second, since Manta Chain was developed fully open source from inception, nearly 1.5 years ago, any attackers would have significantly more time to study and find exploits compared to the length of this audit.

Overall, we assess the Manta Chain codebase to be of very high quality. The documentation provided was detailed and clear and the accompanying test-suites were thorough, exercising all security-critical paths in the codebase.

**Summary of issues detected.** The audit uncovered 19 issues, none of which were assessed to be of high or critical severity by the Veridise auditors. The Veridise auditors also identified 7 issues which were assessed as medium-severity. Two of these issues were due to incorrect

weights set for external calls that could enable a DOS attack (V-MANC-VUL-001 and V-MANC-VUL-003) and one issue was related to an implementation deviation from the MantaPay formal specification (V-MANC-VUL-002).

**Recommendations.** Although none of the issues uncovered in this audit were of high or critical severity, we recommend that the Manta Network developers address the medium-severity issues in the near future. Furthermore, we recommend that Manta Network set up a bug bounty program to incentivize hackers to disclose any vulnerabilities rather than exploit them. Since the codebase has been open source for over 1.5 years, attackers have had a much longer time to find vulnerabilities compared to the length of this audit.

**Disclaimer.** We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

# 2 Project Dashboard

**Table 2.1:** Application Summary.

| Name | Version | Type | Platform |
|---|---|---|---|
| Manta Chain | `45ba60e1d` | Rust | Substrate |

**Table 2.2:** Engagement Summary.

| Dates | Method | Consultants Engaged | Level of Effort |
|---|---|---|---|
| Mar. 6 - April. 17, 2023 | Manual & Tools | 5 | 30 person-weeks |

**Table 2.3:** Vulnerability Summary.

| Name | Number | Resolved |
|---|---|---|
| Critical-Severity Issues | 0 | 0 |
| High-Severity Issues | 0 | 0 |
| Medium-Severity Issues | 7 | 6 |
| Low-Severity Issues | 7 | 7 |
| Warning-Severity Issues | 4 | 3 |
| Informational-Severity Issues | 1 | 0 |
| TOTAL | 19 | 16 |

**Table 2.4:** Category Breakdown.

| Name | Number |
|---|---|
| Logic Error | 7 |
| Bad Extrensic Weight | 4 |
| Consensus | 3 |
| Data Validation | 3 |
| Hash Collision | 1 |
| Maintainability | 1 |

# 3 Audit Goals and Scope

## 3.1 Audit Goals

The engagement was scoped to provide a security assessment of Manta Chain, Manta Network's blockchain implementation. Manta Chain is a Substrate-based Polkadot parachain that provides a protocol called MantaPay where clients (such as other parachains or end-users) can trade and deposit assets privately. The MantaPay protocol consists of two components: an offchain prover which generates zero-knowledge proofs and UTXOs, and an on-chain component which verifies the proofs and updates the ledger with the transactions. Each component consists of a whitepaper and formal specification; this audit was scoped to assess the on-chain component. With this in mind, we sought to answer the following questions in our audit:

- Are there any design flaws with respect to the on-chain part of MantaPay in the whitepaper or formal specification?
- Does the on-chain component of MantaPay adhere to its whitepaper and formal specification?
- Is Manta Chain's delegated proof-of-stake (PoS) implementation correct? In more detail:
  - Are collators in the network properly incentivized to produce high quality blocks?
  - Is it resistant to known PoS attacks like equivocation attacks?
  - Is it possible for delegators to abuse the staking mechanism by either staking too little (or none) or stealing funds?
- Can the external calls exposed by the Manta Chain runtime be used to compromise the security of the system? Specifically:
  - Are the call parameters properly validated?
  - Is the proper authentication/authorization in place for the calls?
- Are there any transactions exposed by Manta Chain that are unsigned which should be signed?
- Are appropriate weights set for all external calls? In particular:
  - Is there high quality benchmarking in place for deriving weights?
  - Does the weight function correspond to the runtime complexity of the external call?
  - Does the weight function account for all storage reads and writes?
  - Finally, can the weight function ever be 0, a case that can admit denial of service bugs?
- Are there any arithmetic overflows/underflows and if so, what are their security impact?

## 3.2 Audit Methodology & Scope

**Audit Methodology.** To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* We used `cargo-audit`, an open-source static analysis tool used to audit Cargo.lock files for crates with security vulnerabilities reported to the RustSec Advisory Database.
- *Fuzzing/Property-based Testing.* We leveraged fuzz testing to determine if Manta Chain's implementation deviated from the intended behavior. To do this, we first encoded *invariants*, logical formulas that should hold throughout Manta Chain's lifecycle, as assertions. We then wrote harnesses for afl.rs, which generated random sequences of external calls relevant to those assertions. If afl.rs found a crash then this indicated either a panic or a violation of the invariant. We provide a table outlining the invariants we fuzzed tested as well as the fuzzing methodology in Chapter 5.

*Scope.* The scope of the audit was limited to all the code in the following top-level directories of Manta Chain:

- `node/*` - starts up a node in Manta Chain
- `pallets/*` - includes the following pallets whose name explains their behavior
  - `asset-manager`
  - `collator-selection`
  - `manta-pay`
  - `parachain-staking`
  - `tx-pause`
  - `vesting`
- `primitives/*` - defines traits and types used by other packages.
- `runtime/*` - contains runtime configuration for different execution environments.

*Methodology.* Veridise auditors first reviewed previous audit reports of substrate blockchains, the documentation provided by Manta Network developers, and inspected the provided tests to determine what logic had been extensively tested. They then began a manual audit of the code assisted by both static analyzers and property-based fuzz testing. During the audit, the Veridise auditors met with the Manta Network developers on a weekly basis and messaged over Telegram to ask questions about the code, and report suspected bugs.

## 3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

**Table 3.1:** Severity Breakdown.

| | Somewhat Bad | Bad | Very Bad | Protocol Breaking |
|---|---|---|---|---|
| Not Likely | Info | Warning | Low | Medium |
| Likely | Warning | Low | Medium | High |
| Very Likely | Low | Medium | High | Critical |

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

**Table 3.2:** Likelihood Breakdown

| | |
|---|---|
| Not Likely | A small set of users must make a specific mistake |
| Likely | Requires a complex series of steps by almost any user(s)<br>- OR -<br>Requires a small set of users to perform an action |
| Very Likely | Can be easily performed by almost anyone |

**Table 3.3:** Impact Breakdown

| | |
|---|---|
| Somewhat Bad | Inconveniences a small number of users and can be fixed by the user |
| Bad | Affects a large number of people and can be fixed by the user<br>- OR -<br>Affects a very small number of people and requires aid to fix |
| Very Bad | Affects a large number of people and requires aid to fix<br>- OR -<br>Disrupts the intended behavior of the protocol for a small group of users through no fault of their own |
| Protocol Breaking | Disrupts the intended behavior of the protocol for a large group of users through no fault of their own |

# 4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

**Table 4.1:** Summary of Discovered Vulnerabilities.

| ID | Description | Severity | Status |
|---|---|---|---|
| V-MANC-VUL-001 | Static fee charged despite dynamic storage access | Medium | Acknowledged |
| V-MANC-VUL-002 | Users can use any previously seen Merkle root | Medium | Acknowledged |
| V-MANC-VUL-003 | MantaPay weights calculated with a small database | Medium | Acknowledged |
| V-MANC-VUL-004 | Total supply of native assets can exceed the set limit | Medium | Acknowledged |
| V-MANC-VUL-005 | Missing updates in update_asset_metadata | Medium | Acknowledged |
| V-MANC-VUL-006 | No slashing mechanism for collators | Medium | Acknowledged |
| V-MANC-VUL-007 | Collators given full rewards regardless of quality | Medium | Open |
| V-MANC-VUL-008 | Missing validation in pull_ledger_diff | Low | Acknowledged |
| V-MANC-VUL-009 | increase_count_of_associated_assets can overflow | Low | Acknowledged |
| V-MANC-VUL-010 | Account checks are incorrect | Low | Acknowledged |
| V-MANC-VUL-011 | Unstaked user may be selected as collator | Low | Acknowledged |
| V-MANC-VUL-012 | XCM instructions can charge 0 weight | Low | Acknowledged |
| V-MANC-VUL-013 | Missing validation in set_units_per_second | Low | Acknowledged |
| V-MANC-VUL-014 | Collator is a single point of failure for a round | Low | Acknowledged |
| V-MANC-VUL-015 | Unchecked index calculation in spend_all | Warning | Open |
| V-MANC-VUL-016 | Excess fees not refunded | Warning | Intended Behavior |
| V-MANC-VUL-017 | Assets can be registered at unsupported locations | Warning | Acknowledged |
| V-MANC-VUL-018 | Minimum delegator funds is not MinDelegatorStk | Warning | Acknowledged |
| V-MANC-VUL-019 | Unintended test crashes | Info | Open |

## 4.1 Detailed Description of Issues

### 4.1.1 V-MANC-VUL-001: Static fee charged despite dynamic storage accesses

| Severity | Medium | Commit | 45ba60e1d |
|---|---|---|---|
| Type | Bad Extrinsic Weight | Status | Acknowledged |
| File(s) | `parachain-staking/lib.rs` | | |
| Location(s) | go_online, go_offline, candidate_bond_more | | |

Blockchain computations must have appropriate fees to prevent network congestion. For substrate extrinsics, these fees are set by computing an associated weight for the operation where the weight is intended to capture the maximum computational cost. As reads from and writes to storage are expensive, these weights should consider the number of these operations that are performed. The following extrinsics, however, have a fixed weight despite requiring a dynamic number of reads or writes due to `insert` or `remove` operations being performed on `CandidatePool`.

- go_online
- go_offline
- candidate_bond_more
- execute_candidate_bond_less
- delegate
- execute_leave_delegators
- delegator_bond_more
- execute_delegation_request
- schedule_leave_delegators
- schedule_delegator_bond_less
- cancel_leave_delegators

Also note that similar functions in the same pallet, such as `schedule_leave_candidates` charge the users dynamic fees. An example can be seen in Snippet 4.1.

**Impact** As the size of the `CandidatePool` grows, the cost of `insert` and `remove` will increase linearly since vector inserts in Rust are linear in the size of the vector. This allows malicious actors to add many candidates to the pool for a fixed monetary cost despite an increasing computational cost. If the size of the pool becomes too large, this could effectively create a DoS.

**Recommendation** Similar to `schedule_leave_candidates`, calculate the weights dynamically rather than charging a fixed cost.

**Developer Response** The developers have acknowledged the issue and are determining how to address it.

```rust
#[pallet::call_index(12)]
#[pallet::weight(<T as Config>::WeightInfo::go_offline())]
/// Temporarily leave the set of collator candidates without unbonding
pub fn go_offline(origin: OriginFor<T>) -> DispatchResultWithPostInfo {
    let collator = ensure_signed(origin)?;
    let mut state = <CandidateInfo<T>>::get(&collator).ok_or(Error::<T>::CandidateDNE
    )?;
    ensure!(state.is_active(), Error::<T>::AlreadyOffline);
    state.go_offline();
    let mut candidates = <CandidatePool<T>>::get();
    if candidates.remove(&Bond::from_owner(collator.clone())) {
        <CandidatePool<T>>::put(candidates);
    }
    <CandidateInfo<T>>::insert(&collator, state);
    Self::deposit_event(Event::CandidateWentOffline {
        candidate: collator,
    });
    Ok(().into())
}
```

**Snippet 4.1:** `go_offline` calls `remove` on the `CandidatePool` but charges users a fixed weight in `WeightInfo::go_offline`

 

### 4.1.2 V-MANC-VUL-002: Users can use any previously seen Merkle root

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 45ba60e1d |
| **Type** | Hash Collision | **Status** | Acknowledged |
| **File(s)** | `pallets/manta-pay/src/lib.rs` | | |
| **Location(s)** | `has_matching_utxo_accumulator_output` | | |

The MantaPay protocol maintains a Merkle tree on the ledger where the leaves of the ledgers are the hashes of the UTXOs generated during the protocol's lifetime. In order to spend a UTXO, users must supply a ZK proof that the UTXO belongs to the Merkle tree on ledger. The membership proof takes as input the root of the Merkle tree (public input), the inner node hashes (private inputs), and proves that root can be derived from the inner node hashes and leaf.

Ideally, the ledger would check that the root provided is equal to the latest root on chain. However, this isn't done in practice as the transaction could easily be front-runned since every transaction changes the root. Instead, the ledger maintains a set of **all** previously generated roots and just checks that the root provided belongs to that set.

However, by allowing the root provided by the user to be any previously generated root, an attacker simply needs to find a hash collision with any previously generated root in order to steal assets. The likelihood of finding a collision grows quadratically with the number of previously seen hashes. In particular, given an output size of $b$ bits and $n$ previously generated hashes, the likelihood of finding a collision with any of the $n$ hashes is is approximately $\frac{n^2}{2^{b+1}}$ .

The current version of the Protocol uses the Poseidon hash function which produces 255 bit hashes and so in theory should be safe even with billions of previously seen roots. However, this is contingent on the safety of the Poseidon hash. While there has been a significant amount of research and analysis conducted on the function, including various attacks and optimizations, there is no formal proof of its security and correctness let alone any proofs about concrete implementations.

**Impact** Storing all previously seen roots significantly increases the likelihood of a collisions. If any attack or weakness is found in the Poseidon hash, then this can be an additional means of attacking the protocol.

**Recommendation** There are a few ways to mitigate this. Protocols like Semaphore maintain a timeout period `TIMEOUT` and associate each root with a timestamp indicating when it was created. Any root created before `now() - TIMEOUT` is rejected. Another option is to only store the `N` previously generated roots and only allow a root if it belongs to the set of `N` previously generated roots. The latter option would have the additional benefit of not needing to store every root on chain.

**Developer Response** The developers acknowledged the issue and will either use a timestamp or only maintain the N previously generated roots.

### 4.1.3 V-MANC-VUL-003: MantaPay weights calculated with a small database

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 45ba60e1d |
| **Type** | Bad Extrinsic Weight | **Status** | Acknowledged |
| **File(s)** | `pallets/manta-pay/src/lib.rs` | | |
| **Location(s)** | `to_private`, `to_public`, `private_transfer` | | |

Transactions `to_public`, `to_private`, and `private_transfer` take as input nullifiers and membership proofs and generate UTXOs. These UTXOs are then added to a Merkle tree on the ledger.

Manta pay shards this Merkle tree into 256 buckets where each bucket has its own Merkle tree. Instead of storing the entire tree at each bucket, the Ledger just stores the last path added to the tree. When adding a UTXO, the Ledger first computes its corresponding bucket, then computes the new path pointing to that UTXO, and finally adds that path to the bucket.

Computing the new path should take time proportional to `log(n)` where `n` is the size of the Merkle Tree. The current benchmarking scheme only covers cases where the previous path is small i.e, at most size 1. However, if the number of transactions gets large i.e, is on the order of hundreds of millions or billions, then the size of the path can get to 24-28 (taking shards into account). If the tree grows to this size, this means each execution of the extrinsic will perform 24-28 hashes, multiplied by the number of UTXOs to be added.

The benchmarking scheme should take into account the size of the tree to make sure that the existing weights are enough to offset the computation of the new Merkle tree path.

**Impact** In general it is important to set the weights to account for both computation and storage; setting the weight too low can allow users to perform a large number of transactions with little cost. In particular, malicious users may take advantage of the low fee to launch a DOS attack.

**Recommendation** There are several ways to address this.

One strategy would be to take in an additional parameter that corresponds to the logarithm of size of the Merkle Tree on the ledger. The weight charged can be proportional to this value. In the implementation, this value (technically $2^{value}$) can be compared against the actual size and the transaction will only proceed if it is larger than or equal to the actual size.

Another strategy would be to benchmark the pallets by taking into account the size of the tree as well. If we expect Manta-Pay to not exceed more than a billion transactions then maybe benchmark the pallet assuming the current path length is around 24-28.

**Developer Response** The developers acknowledged that the weights should be recalculated with a more saturated database.

 

### 4.1.4 V-MANC-VUL-004: Total supply of native assets can exceed the set limit

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 45ba60e1d |
| **Type** | Logic Error | **Status** | Acknowledged |
| **File(s)** | `pallets/asset-manager/src/lib.rs` | | |
| **Location(s)** | `mint_asset` | | |

One invariant underlying the correctness of MantaPay is that the total supply of an asset cannot exceed the maximum amount that can be held in a particular account. This is because Manta-Pay uses a dedicated account `A` to store the value of all the private assets. As such, `A` should, in principle, be able to hold all the supply in the case where all of that asset is privatized.

In more detail, when privatizing a user's public assets (via `to_private`), Manta-Pay constructs opaque utxo's to encode the amount privatized, and then transfers those public assets into `A`. This transfer is expected to not fail because of the invariant described above. However, we found a case where the transfer can fail.

In particular, `Manta` enforces this invariant for `NonNative` assets because every time an asset is minted into an account, the total supply is increased. If the total supply would exceed the maximum that can be held in an account, then the mint fails with the error `Overflow`. However, there is no such check for `Native` assets. As such, if the total supply of `Native` assets exceeds the maximum that can be held in an account, `u128::MAX` , then `to_private` calls that should succeed can fail if the amount held in `A` is close to the maximum allowed. This is demonstrated in Snippet 4.2.

**Impact** By not constraining the amount of Native assets to be less than the maximum amount that can be held in an account, `to_private` transactions that should succeed will fail.

**Recommendation** We recommend a similar check be done for Native assets as is done for NonNative assets to enforce that the total supply cannot exceed the maximum that can be held in a given account.

**Developer Response** The developers have acknowledged the issue and are determining how to address it.

```rust
#[test]
fn public_account_issue() {
    let mut rng = OsRng;
    new_test_ext().execute_with(|| {
        let asset_id = NATIVE_ASSET_ID;
        let value = 1000u128;
        let id = NATIVE_ASSET_ID;
        let metadata = AssetRegistryMetadata {
            metadata: AssetStorageMetadata {
                name: b"Calamari".to_vec(),
                symbol: b"KMA".to_vec(),
                decimals: 12,
                is_frozen: false,
            },
            min_balance: TEST_DEFAULT_ASSET_ED2,
            is_sufficient: true,
        };
        assert_ok!(MantaAssetRegistry::create_asset(
            id, metadata.into(), TEST_DEFAULT_ASSET_ED2,
            true
        ));
        assert_ok!(FungibleLedger::<Test>::deposit_minting(id, &ALICE, 2*value));
        assert_ok!(FungibleLedger::<Test>::deposit_minting(id, &MantaPay::account_id
    (), u128::MAX));


        let mut utxo_accumulator = UtxoAccumulator::new(UTXO_ACCUMULATOR_MODEL.clone
    ());
        let spending_key = rng.gen();
        let address = PARAMETERS.address_from_spending_key(&spending_key);
        let mut authorization =
            Authorization::from_spending_key(&PARAMETERS, &spending_key, &mut rng);
        let asset_0 = Asset::new(Fp::from(asset_id), value);

        // First ToPrivate
        let (to_private_0, pre_sender_0) = ToPrivate::internal_pair(
            &PARAMETERS, &mut authorization.context,
            address, asset_0,
            Default::default(), &mut rng,
        );

        let to_private_0 = to_private_0
        .into_post(
            FullParametersRef::new(&PARAMETERS, utxo_accumulator.model()),
            &PROVING_CONTEXT.to_private,
            None, Vec::new(), &mut rng,
        )
        .expect("Unable to build TO_PRIVATE proof.")
        .expect("Did not match transfer shape.");

        assert_ok!(MantaPay::to_private(
            MockOrigin::signed(ALICE),
            PalletTransferPost::try_from(to_private_0).unwrap()
        ));
    });
}
```

**Snippet 4.2:** Failed `to_private` due to count of native assets exceeding `u128::MAX`

 

### 4.1.5 V-MANC-VUL-005: Missing updates in update_asset_metadata

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 45ba60e1d |
| **Type** | Logic Error | **Status** | Acknowledged |
| **File(s)** | `pallets/asset-manager/src/lib.rs` | | |
| **Location(s)** | `update_asset_metadata` | | |

Manta Chain has an asset-manager pallet which is responsible for registering and minting assets. Each asset has a unique id and is associated with various metadata like a `name`, `symbol`, `decimal places` etc. One important metadata is called `min_balance` . In order to store an account with some quantity of assets on the ledger, it must have more than `min_balance` quantity. This piece of metadata is also used when validating `transfers`.

In particular, many asset transfers take an "existential parameter" as input, called `KeepAlive`, which decides what to do if the transfer would take the account's balance (with respect to the asset) below `min_balance`. If `KeepAlive` is set, then the transfer will fail if the amount goes below the `min_balance`. If it is not set then other configurations come into place and the account may be removed and the remaining balance burned.

The asset-manager pallet exposes an extrinsic called `update_asset_metadata` which takes as input the new metadata for that asset and updates the ledger to associate the asset with that metadata. While the implementation took a new `min_balance` as input, it did not update the ledger to associate the asset with this metadata.

We note the this API also took as input a new value for the metadata `is_sufficient`, but similarly did not update the ledger to associate the asset with this metadata.

**Impact** While it is rare for the `min_balance` to be changed, it is sometimes necessary if it was originally set to high for example. The current API made it appear that `min_balance` could be changed and so users may think the `min_balance` was changed when it fact wasn't.

**Recommendation** The main issue with this extrinsic is its interface makes it appear as though the metadata `min_balance` and `is_sufficient` could be changed when it actually didn't. Either the API should be changed to only take the metadata which should be changed, or it should appropriately update `min_balance` and `is_sufficient`.

**Developer Response** The developers acknowledged this issue and are discussing two possible fixes. The first is to update both parameters in the asset pallet and the second is to change the interface to not allow the `min_balance` or `is_sufficient` parameters to be updated.

### 4.1.6 V-MANC-VUL-006: No slashing mechanism for collators

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 45ba60e1d |
| **Type** | Consensus | **Status** | Acknowledged |
| **File(s)** | parachain-staking | | |
| **Location(s)** | N/A | | |

Proof of Stake blockchains oftentimes have a slashing mechanism to detect poorly performing stakers and punish them. Usually, a large portion of the staker's stake is taken by the chain as punishment for poor performance.

Currently, Manta Chain does not have any slashing mechanism. Instead, it uses a combination of social pressure and manual slashing to incentivize good behavior. In more detail, when the owners detect a poorly performing collator, they will contact the collator over Discord and warn them of the poor performance. If their performance does not improve, the owners will slash the collator's funds manually.

While this may work when the blockchain is small, it will be difficult to enforce as the chain grows. As such, we recommend that Manta Chain put a slashing mechanism in place.

**Impact** Manta Chain's current mechanism of social pressure will only work with a small set of trusted collators. However, as the chain grows, we believe this mechanism is not sufficient for properly incentivizing collators to do a good job.

**Recommendation** We recommend that Manta Chain have a slashing mechanism in place to swap in if/when the current process is insufficient.

**Developer Response** The developers acknowledged that there is no slashing mechanism in place and plan to include one in the future if/when the current process stops working.

 

### 4.1.7 V-MANC-VUL-007: Collators given full rewards regardless of quality

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 45ba60e1d |
| **Type** | Consensus | **Status** | Open |
| **File(s)** | `pallets/parachain-staking/src/lib.rs` | | |
| **Location(s)** | `pay_one_collator_reward` | | |

Manta Chain rewards collators by first allocating a fixed number of points (20) for every block they author and then giving the collator a fixed percentage of those allocated points as rewards. However, there is no check on the quality of the blocks authored by the collator: an empty block will result in just as many rewards as a full block.

Currently, Manta relies on the owners to monitor the blocks on-chain and manually punish collators who perform poorly. However, as the chain grows, this misbehavior may not be easy to detect.

One relatively simple way to address this issue is to adjusting the reward system to incentivize high quality blocks.

**Impact** Collators can effectively steal funds from Manta by authoring low quality blocks (i.e, empty or partial blocks) and reaping full rewards.

**Recommendation** We recommend the developers adjust the rewards system to either reward the collators for high quality blocks or punish them for authoring poor ones.

**Developer Response** TBD

### 4.1.8 V-MANC-VUL-008: Missing validation in pull_ledger_diff

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 45ba60e1d |
| **Type** | Data Validation | **Status** | Acknowledged |
| **File(s)** | `pallets/manta-pay/src/lib.rs` | | |
| **Location(s)** | Line 593 | | |

`pull_ledger_diff` takes as input a `Checkpoint` which is a struct of two fields `receiver_index` and `sender_index` and pulls sender and receiver data from the ledger starting at `sender_index` (resp. `receiver_index`) up till at most `sender_index + PULL_MAX_SENDER_UPDATE_SIZE` (resp. `receiver_index + PULL_MAX_RECEIVER_UPDATE_SIZE`). However, there is no check that this sum cannot overflow for both the sender and receiver index in `pull_senders`, `pull_receivers`, `pull_senders_for_shard` and `pull_receivers_for_shard`.

**Impact** If the code is compiled without `--release` flag then a malicious user could crash the node by passing in bad values. If it is built with `--release` then the call will be reported as successful and no senders or receivers will be returned. However, if a benign end user is calling the API with incorrect indexes it might be better to return an Error informing them that the index is invalid.

**Recommendation** We recommend adding bounds checks to be safe and to return an Error.

**Developer Response** The developers are aware and agree that it would be better to check and return an error.

### 4.1.9 V-MANC-VUL-009: increase_count_of_associated_assets can overflow

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 45ba60e1d |
| **Type** | Logic Error | **Status** | Acknowledged |
| **File(s)** | `pallets/asset-manager/src/lib.rs` | | |
| **Location(s)** | Line 590 | | |

The asset_manager pallet maintains a mapping of paraids to a count of assets associated with that paraid. Each paraid can be associated with at most u32::MAX assets. When registering an asset or moving its location, the pallet calls `increase_count_of_associated_assets` which takes as input a paraid and increments the number of assets associated with that paraid. However, this function does not check whether increasing the number of assets will result in an overflow.

**Impact** If the runtime is compiled using `--debug` then this can crash the node. However, if built under `--release` then the asset count will go to zero.

**Recommendation** Make this function check if the addition will result in an overflow i.e, check if the current count is `u32::MAX` and return an error.

**Developer Response** Acknowledged

### 4.1.10 V-MANC-VUL-010: Account checks are incorrect.

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 45ba60e1d |
| **Type** | Logic Error | **Status** | Acknowledged |
| **File(s)** | `pallets/manta-pay/src/lib.rs` | | |
| **Location(s)** | `check_sink_accounts`, `check_source_accounts` | | |

When validating a transaction, the source and sink accounts are checked by `check_sink_accounts` and `check_source_accounts`. These functions iterate over pairs `(account, value)` and check that `value` can be safely deposited (withdrawn) from `account`. The logic is correct only if every account only appears in at most one pair. While this is fine for the current APIs, if the APIs change to allow multiple sink or multiple source accounts, then this code needs to be refactored or the uniqueness needs to be enforced elsewhere.

**Impact** Currently there is no impact since the current APIs only allow one account for the source and sink accounts.

**Recommendation** To be safe, we recommend you add additional check in the validation step to ensure the accounts are distinct for both sources and sinks.

**Developer Response** The developers acknowledged the issue and plan to add a check during validation that the accounts are distinct.

 

### 4.1.11 V-MANC-VUL-011: Unstaked user may be selected as collator

| Severity | Low | Commit | 45ba60e1d |
|---|---|---|---|
| Type | Logic Error | Status | Acknowledged |
| File(s) | parachain-staking/lib.rs | | |
| Location(s) | select_top_candidates | | |

Parachains use collators to combine transactions into blocks that are then checked by Validators on the relay chain. Notably, this allows collators to remain relatively untrusted as validators will ensure blocks were created correctly. On Manta's chain, collators are selected from a group of staked users who receive rewards for creating blocks. Requiring that collators be staked provides additional security guarantees as if a collator does misbehave (e.g. submit no blocks for validation, submit multiple conflicting blocks), governance can step in and slash the user's staked funds. As such, unstaked collators have less incentive to maintain the stability of the parachain and therefore should be avoided. In the collator selection process though, if no sufficiently staked collator can be found, collators from the previous round will be selected as shown below. As there is no validation as to the current state of the previous collators' stake, this could select completely unstaked validators who have no incentive to ensure network stability. Here is a simple test case which demonstrates this occurring:

```
#[test]
fn test_failed_candidate_selection() {
    ExtBuilder::default()
        .with_balances(vec![(10, 10)])
        .with_candidates(vec![(10, 10)])
        .build()
        .execute_with(|| {
            roll_to(2);
            // Account 10 leaves
            assert_ok!(ParachainStaking::schedule_leave_candidates(
                Origin::signed(10),
                6u32
            ));
            // move to round where we get update
            roll_to(5);
            let candidate: Vec<u64> = ParachainStaking::selected_candidates();
            assert_ne!(candidate[0], 10u64);
        });
}
```

**Impact** Collators will not be incentivized to ensure network stability. As such, it is possible that another set of partially staked or perhaps "trusted" collators would provide better stability.

**Recommendation** The developers may want to consider maintaining a set of "trusted" collators to fall back on in case no staked collators can be found.

```rust
fn select_top_candidates(now: RoundIndex) -> (u32, u32, BalanceOf<T>) {
    let (mut collator_count, mut delegation_count, mut total) =
        (0u32, 0u32, BalanceOf::<T>::zero());
    // choose the top TotalSelected qualified candidates, ordered by stake
    let collators = Self::compute_top_candidates();
    if collators.is_empty() {
        // SELECTION FAILED TO SELECT >=1 COLLATOR => select collators from previous
    round
        let last_round = now.saturating_sub(1u32);
        let mut total_per_candidate: BTreeMap<T::AccountId, BalanceOf<T>> = BTreeMap
    ::new();
        // set this round AtStake to last round AtStake
        for (account, snapshot) in <AtStake<T>>::iter_prefix(last_round) {
            collator_count = collator_count.saturating_add(1u32);
            delegation_count =
                delegation_count.saturating_add(snapshot.delegations.len() as u32);
            total = total.saturating_add(snapshot.total);
            total_per_candidate.insert(account.clone(), snapshot.total);
            <AtStake<T>>::insert(now, account, snapshot);
        }
        // `SelectedCandidates` remains unchanged from last round
        // emit CollatorChosen event for tools that use this event
        for candidate in <SelectedCandidates<T>>::get() {
            let snapshot_total = total_per_candidate
                .get(&candidate)
                .expect("all selected candidates have snapshots");
            Self::deposit_event(Event::CollatorChosen {
                round: now,
                collator_account: candidate,
                total_exposed_amount: *snapshot_total,
            })
        }
        return (collator_count, delegation_count, total);
    }

    ...
}
```

**Snippet 4.3:** Candidate selection code that can select unstaked collators

### 4.1.12 V-MANC-VUL-012: XCM instructions can charge 0 weight

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 45ba60e1d |
| **Type** | Bad Extrinsic Weight | **Status** | Acknowledged |
| **File(s)** | `runtime/(calamari, dolphin)/src/weights/xcm/mod.rs` | | |
| **Location(s)** | Every use of `weigh_multi_assets` | | |

The polkadot ecosystem uses the XCM messaging standard to enable parachains and the relay-chain to communicate with each other. For example, if a parachain P1 wants to deposit an asset onto another parachain P2 they can construct an XCM message saying they wish to deposit an asset into an account associated with P1 and send it to P2.

In more detail, each XCM message consists of a sequence of low level XCM instructions that get executed by the `XCM executor` on the destination parachain. To offset the cost of executing these instructions, parachains are responsible for setting weights for each instruction. That way, the sender of the XCM can be charged fees for the destination parachain executing their message.

Manta chain configured the weights of multiple instructions in such a way that senders could generate messages that totaled 0 weight. For example, here is the code snippet which sets the weight for `deposit_asset`:

```
fn deposit_asset(
        assets: &MultiAssetFilter,
        _max_assets: &u32,
        _dest: &MultiLocation,
    ) -> Weight {
        // Hardcoded until better understanding how to deal with worst case scenario
    of holding register
        let hardcoded_weight: u64 = 1_000_000_000;
        let weight = assets.weigh_multi_assets(XcmFungibleWeight::<Runtime>::
    deposit_asset());
        cmp::min(hardcoded_weight, weight)
    }
```

Here, `deposit_asset` sets the weight for the XCM instruction `deposit_asset` which takes as input a parameter called `assets`. For simplicity, we can think of `assets` as a vector of assets. This function sets the weight to be the minimum of a hard coded weight and the result of `weight_multi_assets` which returns `0` when the length of `assets` is 0. Thus, if `deposit_asset` is called with an empty vector of `assets`, then the instruction has weight 0 and the caller is not charged.

This may allow malicious or incompetent senders the ability to spam Manta since the cost for sending the message is 0 even though the instruction will get successfully executed by the XCM executor. In general, setting weights to 0 can lead to a denial of service, however, in this case a denial of service might be difficult since when the instruction is invoked on a vector of length 0, the execution is very fast. However, to avoid spam and incompetent usage we recommend that Manta add a minimal base fee since for instructions that can be executed with 0 weight.

**Impact** Malicious users may be able to spam Manta with XCM messages of weight 0. This spam could slow down the performance of the blockchain and potentially result in a denial of

service.

**Recommendation** We recommend that a base fee always be charged to prevent spam.

**Developer Response** The developers acknowledged the issue and decided to change `weigh_multi_assets` to charge the benchmarked weight of of a single asset execution for any successfully executed multiasset XCM message.

### 4.1.13 V-MANC-VUL-013: Missing validation in set_units_per_second

| Severity | Low | Commit | 45ba60e1d |
|---|---|---|---|
| Type | Data Validation | Status | Acknowledged |
| File(s) | pallets/asset-manager/src/lib.rs | | |
| Location(s) | set_units_per_second | | |

The `asset-manager` pallet manages a hashmap called `UnitsPerSecond` which maps `assetIds` to a `u128` value `units_per_second` which is used to determine the price to perform an XCM transfer. It exposes a function called `set_units_per_second` which can be used to set the `units_per_second` for a given asset. The `units_per_second` value is used to determine the cost (in terms of the corresponding asset) of purchasing a given weight to perform a transaction. The code snippet which determines the cost is shown below:

```
let units_per_second = M::units_per_second(&asset_id).ok_or({
                    log::debug!(
                        target: "FirstAssetTrader::buy_weight",
                        "units_per_second missing for asset with id: {:?}",
                        id,
                    );
                    XcmError::TooExpensive
                })?;

let amount = units_per_second * (weight as u128) / (WEIGHT_PER_SECOND as u128);
// we don't need to proceed if amount is zero.
// This is very useful in tests.
if amount.is_zero() {
    return Ok(payment);
}
let required = MultiAsset {
    fun: Fungibility::Fungible(amount),
    id: XcmAssetId::Concrete(id.clone()),
};
```

It calculates `amount` using the multiplication operator $*$ which can overflow. Currently, `units_per_second` and `amount` are of type `u128` and if `units_per_second` is larger than `u128::MAX / (u64::MAX)` then someone can purchase a large amount of weight i.e, `u64::MAX` for a small amount of a given asset. This can allow a malicious parachain to perform a DOS attack on the chain.

Currently there is no validation in `set_units_per_second` on the parameters to ensure the `units_per_second` is sufficiently small. However, since `set_units_per_second` can only be called by the `root` , this is unlikely to occur. Nevertheless, if the root user sets this accidentally or is tricked into setting an excessively large value then this attack is possible.

**Impact** If `units_per_second` is set larger than `u128::MAX / (u64::MAX)` then someone can purchase large amounts of weight at a low cost, which can lead to a DOS attack on the chain.

**Recommendation** We recommend either changing the type of the storage variable holding `units_per_second` to be a map of assetId to a value of type `u64` or by validating that the amount is sufficiently small.

**Developer Response** The developers acknowledged the issue and will fix the weight calculation to use saturating arithmetic. That should address this issue.

 

### 4.1.14 V-MANC-VUL-014: Collator is a single point of failure for a round

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 45ba60e1d |
| **Type** | Consensus | **Status** | Acknowledged |
| **File(s)** | | N/A | |
| **Location(s)** | | N/A | |

The Manta parachain uses the Aura consensus mechanism to select collators to author blocks. Aura selects a primary collator for a round and only that collator is allowed to produce blocks in that round. However, if that collator goes down then no blocks will get produced which makes that collator a single point of failure.

Other parachains like Moonbeam address this by selecting multiple collators in a given round.

**Impact** If a collator goes down, then no blocks will get produced for a given round, thereby impacting the transaction throughput of Manta.

**Recommendation** We recommend that Manta use a consensus mechanism that selects multiple collators. Ideally, this mechanism would choose geographically separated collators so if one collator goes down the likelihood of the other going down is low.

**Developer Response** The developers are aware of this issue and have plans in place to move away from the Aura consensus mechanism.

### 4.1.15 V-MANC-VUL-015: Unchecked index calculation in spend_all

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 45ba60e1d |
| **Type** | Logic Error | **Status** | Acknowledged |
| **File(s)** | `pallets/manta-pay/src/lib.rs` | | |
| **Location(s)** | `spend_all` | | |

The `spend_all` function in the Manta-Pay pallet does the following:

1. Adds the nullifier commitments in the TransactionPost to the `NulliferCommitmentSet`
2. Inserts each `(nullifier, outgoingNote)` pair to the `NullifierSetInsertionOrder` structure.
3. Updates a global variable `NullifierSetSize` which stores the size of the nullifier commitment set.

The index where the pair gets inserted, along with the new nullifier size, is based on a calculation `index + i` where `i` is the index of the corresponding SenderPost and `index` is the current size of the set. However, this arithmetic is unchecked and could result in an overflow.

**Impact** When the size of the commitment set is `u64::MAX`, the computation for the index to insert overflows which results in the pair getting inserted at the beginning of the list. Furthermore, the size of the nullifier set is also set to 1. However, this is very unlikely to occur as this value is extremely large and will not be reached through normal execution.

**Recommendation** Add an overflow check and return an error.

**Developer Response** TBD

### 4.1.16 V-MANC-VUL-016: Excess fees not refunded

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 45ba60e1d |
| **Type** | Bad Extrinsic Weight | **Status** | Intended Behavior |
| **File(s)** | parachain-staking/lib.rs | | |
| **Location(s)** | (cancel_leave, execute_leave, schedule_leave, join)_candidates | | |

When a substrate extrinsic is created, its weight must be carefully considered to ensure it correctly reflects the computational cost of the operation as extrinsic weight is directly related to the fees that are charged to the user. This weight should capture the maximum number of computational resources that will be consumed by the extrinsic as excess fees can be returned. In several functions, though, the weights are computed based on the value of an argument provided by the user which might not always reflect the true cost of the computation. For example, consider the following:

```
#[pallet::call_index(11)]
#[pallet::weight(<T as Config>::WeightInfo::cancel_leave_candidates(*candidate_count)
    )]
/// Cancel open request to leave candidates
/// - only callable by collator account
/// - result upon successful call is the candidate is active in the candidate pool
pub fn cancel_leave_candidates(
    origin: OriginFor<T>,
    #[pallet::compact] candidate_count: u32,
) -> DispatchResultWithPostInfo {
    ...

    let mut candidates = <CandidatePool<T>>::get();
    ensure!(
        candidates.0.len() as u32 <= candidate_count,
        Error::<T>::TooLowCandidateCountWeightHintCancelLeaveCandidates
    );

    ...
    Ok(().into())
}
```

In this function, the weight is computed using the `candidate_count` argument, and in order for the function to execute successfully, `candidate_count` must be greater than or equal to the current size of the candidate pool. A user might need to call this function with a `candiate_count` that is larger than the size of the pool to prevent a front-running attack where a malicious user would add candidates to prevent the transaction from executing successfully. In such a case, the weight would be larger than necessary, but no fees are returned to the user.

**Impact** Such functions can charge unnecessary fees to the user.

**Recommendation** Refund the user additional fees that are not consumed.

**Developer Response** Since these extrinsics are likely to be executed sparingly and since the additional fees are likely to be small, we feel like the additional computational cost of determining the excess does not.

### 4.1.17 V-MANC-VUL-017: Assets can be registered at unsupported locations

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 45ba60e1d |
| **Type** | Data Validation | **Status** | Acknowledged |
| **File(s)** | `pallets/asset-manager/src/lib.rs` | | |
| **Location(s)** | `register_asset` | | |

The asset-manager pallet allows assets to be registered, managed, and minted. In particular, `register_asset` takes as input an `asset`, `location`, and corresponding `asset_metadata` and register the asset. Every asset must be associated with a location; however, Manta only supports assets from specific locations. The current implementation of `asset-manager` does not perform any validation on the locations passed into `register_asset` potentially allowing assets to be registered from untested locations. The pallet also exposes a method called `update_asset_location` which is supposed to update the location of an asset. It similarly does not perform any validation on the new location of the asset.

**Impact** The current implementation allows assets to be registered from untested locations.

**Recommendation** The `asset-manager` pallet already implements the `Contains` trait which exposes a method `contains` which takes as input a location and returns true if and only if the location is supported. Currently that method is unused and can be used to validate the locations passed in.

**Developer Response** The developers acknowledged the issue and plan to add a check in `register_asset`.

### 4.1.18 V-MANC-VUL-018: Minimum delegator funds is not MinDelegatorStk

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 45ba60e1d |
| **Type** | Logic Error | **Status** | Acknowledged |
| **File(s)** | `parachain-staking/lib.rs` | | |
| **Location(s)** | N/A | | |

In the case where `MinDelegation < MinDelegatorStk`, it is possible for the delegator's staked funds to be less than `MinDelegatorStk`. This can occur through the following sequence of calls:

1. `delegate` amount `N` from delegator `D` to candidate `C1` where `N >= MinDelegatorStk`
2. `delegate` amount `M` from delegator `D` to candidate `C2` where `M < MinDelegatorStk` and `M >= MinDelegation`
3. `schedule_leave_candidates` and `execute_leave_candidates` for `C1`

This results in `D` having `M` funds staked, where `M < MinDelegatorStk`.

**Impact** If `MinDelegation` is less than `MinDelegatorStk`, a delegator end up with less than `MinDelegatorStk` funds actually staked.

Note that this is not currently exploitable because `MinDelegation == MinDelegatorStk` in all production runtimes. However, if these values are adjusted in the future, this bug may become exploitable.

**Recommendation** There are two options

1. When starting a runtime, ensure that `MinDelegation >= MinDelegatorStk`
2. Whenever a delegation is removed (such as in `execute_leave_candidates`), ensure that the remaining locked funds for the delegator are at least `MinDelegatorStk`.

**Developer Response** The developers acknowledged the issue and are considering removing MinDelegation as there is no apparent reason for it being different from MinDelegatorStk.

### 4.1.19 V-MANC-VUL-019: Unintended test crashes

| Severity | Info | Commit | 45ba60e1d |
|---|---|---|---|
| Type | Maintainability | Status | Open |
| File(s) | pallets/manta-pay/src/lib/rs | | |
| Location(s) | to_private_should_work | | |

Many of the manta-pay tests randomly generate an asset id, total supply, and an amount to make private. To ensure the total supply of the asset is greater than the minimum balance, the minimum balance is always added to randomly generated total supply as seen in this test:

```
fn to_private_should_work() {
    let mut rng = OsRng;
    for _ in 0..RANDOMIZED_TESTS_ITERATIONS {
        new_test_ext().execute_with(|| {
            let asset_id = rng.gen();
            let total_free_supply = rng.gen();
            initialize_test(asset_id, total_free_supply + TEST_DEFAULT_ASSET_ED);
            mint_private_tokens(
                asset_id,
                &value_distribution(5, total_free_supply, &mut rng),
                &mut rng,
            );
        });
    }
}
```

If the random number generator generates a value for the `total_free_supply` which is greater than `u128::MAX - TEST_DEFAULT_ASSET_ED` then the test will fail even though it is expected to succeed.

**Impact** May cause tests to fail when they are expected to succeed.

**Recommendation** Change the test to generate a value for `total_free_supply` between `[0, u128::MAX - TEST_DEFAULT_ASSET_ED)`

**Developer Response** TBD

# 5 Fuzz Testing

## 5.1 Methodology

Our goal was to fuzz test Manta Chain to assess its functional correctness i.e, whether the implementation deviates from the intended behavior. We used afl.rs as our fuzzer and started writing invariants –logical formulas that should hold after every transaction. We then encoded those invariants as assertions in Rust. For each invariant, we wrote a harness which executed a random sequence of relevant external calls and then asserted the invariant should hold after the calls. We prioritized invariants which had a higher security impact e.g, if violated would allow someone to steal funds. For all invariants, we ran afl.rs for at least 24 hours for each invariant.

## 5.2 Properties Fuzzed

The following table describes the invariants we fuzz-tested. The first column states which subsystem (e.g, pallet or xcm) the invarianted is associated with. The second describes the invariant informally in English and the last column notes whether we found a bug when fuzzing the invariant (✗ indicates no bug was found and ✓ means fuzzing this invariant revealed a bug). We ran afl.rs for 24 hours when fuzz-testing each invariant. In the table we use the term “Private transactions" to refer to the collection of external calls in the manta-pay pallet that take a TransferPost; namely, `to_private`, `to_public` and `private_transfer`.

**Table 5.1:** Invariants Fuzzed.

| Subsystem | Invariant | Bug |
|---|---|---|
| para-staking | No account is both a collator and delegator | ✗ |
| para-staking | No account appears in the candidate pool more than once | ✗ |
| para-staking | No account appears as a key in DelegatorStake more than once | ✗ |
| Any | When external calls fail, the state should be unaffected | ✗ |
| para-staking | Candidate bond is at most their free balance and at least minimum stake | ✗ |
| para-staking | At the start of every round, SelectedCandidatePool is a subset of CandidatePool | ✗ |
| para-staking | All candidate accounts have associated info | ✗ |
| para-staking | All candidate collators have an active state | ✗ |
| para-staking | Candidate bond is at most delegator's free balance and at least minimum stake | ✗ |
| para-staking | A delegator can only have one delegation per candidate | ✗ |
| para-staking | A candidate's TopDelegations and BottomDelegations are sorted | ✗ |
| para-staking | For each cand., the lowest top delegation amount is larger than the greatest bottom | ✗ |
| para-staking | Any delegation must be larger than MinDelegation and already staked MinDelegatorStake | ✗ |
| para-staking | A delegator must always stake at least MinDelegatorStake | ✓ |
| para-staking | If a delegator is in TopDelegations or BottomDelegations they must have registered a bond | ✗ |
| para-staking | If a delegator is in TopDelegations or BottomDelegations they must have registered a bond | ✗ |
| asset-manager | AssetIdMetadata and Assets should point to the same metadata values for any asset id | ✓ |
| asset-manager | Minting an asset for any beneficiary should accurately update the balance | ✗ |
| asset-manager | Minting or burning an asset shouldn't allow an account's balance to go below minBalance | ✗ |
| xcm | If xcm fee is less than MinXcmFee, transfer should not succeed | ✓ |
| xcm | Transfer does not change total supply and balance is calculated correctly | ✗ |
| xcm | Xcm instructions are filtered correctly based on the xcm_config filtering | ✗ |
| xcm | buy_execution does not accept non-fungible assets | ✗ |
| xcm | buy_execution correctly calculates the purchase and refund amount | ✗ |
| manta-pay | For all `u128` values $u$, `fp_encode(fp_decode(`$u$`))` = $u$ | ✗ |
| manta-pay | For all ZK proofs $p$ on the subgroup of Bn254, `proof_encode(proof_decode(`$p$`))` = $u$ | ✗ |
| manta-pay | Private transactions should throw error when given a random proof | ✗ |
| manta-pay | Given a transfer post that should succeed, changing the ZK proof should result in an error | ✗ |
| manta-pay | Mutating the public inputs for a valid TransferPost should produce an error | ✗ |
| manta-pay | `pull_ledger_diff` should always succeed or throw an error | ✓ |
| manta-pay | Ledger should always throw an error when given an invalid signature | ✗ |
| manta-pay | Ledger should always throw an error for a transfer with insufficient balance | ✗ |
| manta-pay | Transactions should never allow an account's balance to go below `min_balance` | ✗ |
| manta-pay | Transactions should never allow an account's balance to go below `min_balance` | ✗ |
| manta-pay | UTXO accumulator in TransferPost should equal a "current" accumulator on-chain | ✓ |
| manta-pay | No private transaction can succeed if the nullifier exists on the Ledger | ✗ |
| manta-pay | No private transaction can succeed if a nullifier appears twice in the TransferPost. | ✗ |
| manta-pay | Private transactions succeed only if the UTXOs in the receiver of the TransferPost are distinct | ✗ |
| manta-pay | `to_private` should always succeed if the amount privatized is sufficiently small | ✓ |