Security Audit Report

# Q2 2025 Namada: e2e Shielded Transaction & Balance Consistency

Last Revised
2025/07/24

Authors:
Manuel Bravo, Aleksandar Stojanovic, Ivan Golubovic, Tatjana Kirda

# Contents

# Audit overview

## The Project

From May 2025 to June 2025, the Anoma Foundation engaged Informal Systems to work on a partnership and conduct a security audit. The audit was concerned with the end-to-end execution of shielded transactions with a focus on the consistency of user balances.

## Scope of this report

The scope includes the following main items from the Namada codebase↗:

- Shielded sync
  - `crates/apps_lib/src/client/masp.rs`. Main function: `syncing`
- Transaction construction
  - `crates/apps_lib/src/cli/client.rs`. Main functions: `TxShieldedTransfer`, `TxShieldingTransfer`, `TxUnshieldingTransfer`
  - `crates/apps_lib/src/client/tx.rs`. Main functions: `submit_shielded_transfer`, `submit_shielding_transfer`, `submit_unshielding_transfer` and `submit_ibc_transfer`
  - `crates/sdk/src/tx.rs`. Main functions: `build_shielded_transfer`, `build_shielding_transfer`, and `build_unshielding_transfer`
  - `crates/shielded_token/src/masp/shielded_wallet.rs`. Main function: `gen_shielded_transfer`
  - `masp_primitives-1.4.0/src/transaction/builder.rs`. Main function: `build`
  - `crates/sdk/src/tx.rs`. Main functions: `build, prepare_tx`, `process_tx`, `broadcast_tx`, and `submit_tx`
- Transaction execution:
  - `crates/node/src/shell/prepare_proposal.rs`
  - `crates/node/src/shell/process_proposal.rs`
  - `crates/node/src/shell/finalize_block.rs`. Main functions: `finalize_block`, `retrieve_and_execute_transactions`, and `execute_tx_batches`
  - `crates/node/src/protocol.rs`. Main functions: `apply_wrapper_tx`, `dispatch_tx`, `dispatch_inner_txs`, `apply_wasm_tx`, and `execute_tx`
  - `crates/vm/src/wasm/run.rs`. Main function: `tx`
  - `wasm/tx_transfer/src/lib.rs`. Main function: `apply_tx`
    - → Other related functions such as `token::multi_transfer`, `apply_transparent_transfers`, `multi_transfer`, and `apply_shielded_transfer`were also included in scope.
  - `crates/token/src/tx.rs`. Main functions: `multi_transfer`, and `apply_shielded_transfer`
- Transaction validation:
  - `crates/node/src/protocol.rs`. Main functions: `check_vps`, and `execute_vps`
  - MASP VP: `crates/shielded_token/src/vp.rs`
  - Multitoken VP: `crates/trans_token/src/vp.rs`

The code in scope was audited at the `3531f980fa5be4274d5b50f52e165f3b6b2882db` commit hash.

Notably, the MASP code↗ was out of the scope of this audit, and the zk-related verification logic was not inspected.

## Audit plan

The audit was conducted between May 27, 2025 and June 24, 2025 by the following personnel:

- Ivan Golubovic
- Tatjana Kirda
- Aleksandar Stojanovic
- Manuel Bravo

## Conclusions

No critical issues were identified within the defined scope of this audit. However, two medium severity issues were found in components outside the audit scope, specifically within the `process_proposal` and `prepare_proposal` code paths.

In total, six findings were reported: two classified as medium severity, two as low severity, and two as informational. Detailed information on each issue is provided on the Findings page.

Notably, validity predicates assume that only predefined, whitelisted transactions are executed, introducing implicit dependencies on specific transaction execution logic. While transaction types are restricted via allowlists (code ref ↗), VPs assume that certain invariants hold based on the expected transaction implementation. These include, for example, source and target accounts balance changes being enforced accurately by the transfer WASM code (code ref ↗), and the correct population of the `debited_accounts` (code ref ↗).

# Audit Dashboard

## Target Summary

- **Type:** Implementation
- **Platform:** Rust
- **Artifacts:** Namada repository ↗

## Engagement Summary

- **Dates**: 27.05.2025 - 24.06.2025
- **Method**: Manual code review

## Severity Summary

| Finding Severity | Number |
|---|---|
| Critical | 0 |
| High | 0 |
| Medium | 2 |
| Low | 2 |
| Informational | 2 |
| **Total** | 6 |

# System Overview

We include a set of definitions and assumptions that will be used in the threat model. The properties in the threat model define the desired behavior of the components under scope.

## Definitions

A transfer command is one of the following cli commands: `TxShieldedTransfer`, `TxShieldingTransfer`, `TxUnshieldingTransfer` and includes at least the following arguments:

- A set of sources. Each is composed by:
  - source is an extended spending key or a Namada address
  - token address
  - amount is any quantity
- A set of targets. Each is composed by:
  - target is a shielded payment address or a Namada address
  - token address
  - amount is any quantity
- Fee payer defines the fee payer. It can be an extended spending key if the fee is paid with tokens managed by MASP, or a Namada address if not.
- Fee token is the token in which the fee will be paid by the fee payer.
- Fee amount is the amount of fee tokens that the fee payer is willing to pay per gas unit.
- Gas limit is the maximum amount of gas that the fee payer is willing to pay.

A transfer transaction is the result of executing a transfer command. Their builders define its validity and well-formedness. It includes the following arguments:

- A transparent bundle with a set of transparent inputs (vin) and outputs (vout). A transparent input includes an asset, amount, and source address. A transparent output includes an asset, amount, and target address.
- A sapling bundle with shielded spends, converts, and outputs descriptions.
- Fee payer, fee token, fee amount, and gas limit.

Four transaction states:

- Submitted: A transaction is considered submitted once it is being successfully created. It is the transaction's initial state.
- Accepted: A transaction becomes accepted when it is included in a block proposal that is accepted by the validators.
- Executed: A transaction becomes executed after its execution in finalize block.
- Validated: A transaction becomes validated after its validation in finalize block. If the transaction passes validation, we say that the transaction is committed, i.e., its state changes are persisted. If the transaction fails validation, we say that the transaction is rejected, i.e., its state changes are discarded.

## Assumptions

Assumption 1: A client always executes shielded synchronization before submitting a shielded, shielding, or unshielding command.

Assumption 2: A submitted transaction is eventually included in a block proposal of an honest proposer under

good network conditions, i.e., if validators accepts the proposal via processProposal, the block will be committed.

Assumption 3: The MASP rewards inflation is well computed, and sufficient funds are minted at the MASP address.

# Threat Model

## Property 01: The transaction builders guarantee at execution time that (i) the fee amount of the fee token is greater than or equal to the minimum fee amount, and (ii) the fee payer has enough balance

### Violation consequences

If there is no check and the user picks a fee amount that it is smaller than the fee amount, it is then likely that the transaction is never included in a block. This would pose a liveness issue. Missing the balance check would pose a similar problem.

### Threats

- Threat 1.1. There is no code checking that the fee amount is greater or equal to the minimum fee amount.
- Threat 1.2. There is no code checking that the balance of the fee payer is greater or equal to the total fee.

### Conclusion

The property does not hold. The main reason is that any blockchain state that the client uses to validate the transaction's input may be outdated at the time of the transaction's execution. This is a fundamental issue that cannot be overcome. Given this impossibility, we check a weaker property:

Given a blockchain state fetched from an honest node, the builders check that based on the latest fetched blockchain state (i) the user's proposed fee amount is greater than or equal to the minimum fee amount, and (ii) the fee payer has enough balance to pay for the transaction's fee.

We now provide conclusions for each of the threats.

#### Threat 1.1. conclusion

***The threat is not applicable.***

The builders validate fee-related data in the `validate_fee` function. Let `minimum_fee` be the minimum amount retrieved from the node and `fee_amount` the one provided by the user as input. If the user does not set the force flag to true (the relevant case), the function returns either an error, e.g., if there is no minimum fee for that token in the fetched state, or the maximum between `minimum_fee` and `fee_amount`.

The function is used in the three transaction builders in scope: `build_shielded_transfer`, `build_shielding_transfer` and `build_unshielding_transfer`. Furthermore, the value returned is used by the transaction builder at the end of this process to set the transaction's fee amount, which ensures the required.

#### Threat 1.2. conclusion

***The threat is not applicable.***

The `build_shielding_transfer` builder validates the fee payer balance in `validate_transparent_fee`. The function returns an error if the fee payer does not have enough balance to pay the fees, unless in force mode.

The `build_shielded_transfer` and `build_unshielding_transfer` builders validate the fee payer's balance either in `get_masp_fee_payment_amount` is the fees are paid from a transparent address or implicitely when they compute shielded inputs (in `compute_change`) if the fees are paid from MASP.

## Property 02: Fee payer has sufficient funds and fee meets minimum requirements during prepare proposal

Let t be a transaction included in a block proposal b. An honest proposer includes t in a proposal via `prepare_proposal` if

- the fee payer is guaranteed to have sufficient funds to cover the transaction execution, i.e., the fee payer balance of the fee token during the transaction's execution is greater than or equal to (gas limit * fee amount)
- the fee amount is greater than or equal to the minimum fee amount for the given token during the transaction's execution.

### Violation consequences

- Transactions with insufficient fee payer balances or inadequate fee amounts may be included in block proposals, leading to failed fee transfers during block execution. This results in transaction failures, wasted computational resources, and potential economic losses for validators who cannot collect fees from failed transactions.

### Threats

- Threat 2.1. There is no code checking that the fee payer has sufficient funds.
- Threat 2.2. There is code checking that the fee payer has sufficient funds, but it does not consider that the fee payer may also be the fee payer of some preceding transactions in the same block.
- Threat 2.3. There is no code checking that the fee amount is greater than or equal to the minimum fee amount for the given token.
- Threat 2.4. There is code checking that the fee amount is greater than or equal to the minimum fee amount for the given token but it does not consider that the minimum can be updated via governance.

#### Conclusion

The property holds.

#### Threat 2.1. conclusion

***The threat is not applicable.***

Check for sufficient funds is done in the `transfer_fee` function (ref↗). The function reads the fee payer's balance and uses `checked_sub` to verify sufficient funds before attempting the transfer. If insufficient funds are detected, MASP fee payment (ref↗) is attempted by executing the first transaction of the batch to unshield funds, then balance check is done again. If the fee payer still has insufficient funds after the MASP payment attempt, the transaction is rejected.

#### Threat 2.2. conclusion

***The threat is not applicable.***

The temporary state mechanism in `prepare_proposal` (ref↗) creates an isolated environment with its own write log for transaction simulation. Using this mechanism mitigates threat because when simulating transactions, each transaction's effects, including fee payments, are accumulated in the temporary write log. All the balance changes from all preceding transactions are considered, as those changes are reflected in the write log.

#### Threat 2.3. conclusion

***The threat is not applicable.***

Minimum gas price is determined (ref↗) by comparing consensus-mandated minimum and proposer's own minimum, using the higher value. The `fee_data_check` (ref↗) ensures the fee amount per gas unit is greater than or equal to the minimum gas price for the given token.

#### Threat 2.4. conclusion

***The threat is not applicable.***

While the governance can update minimum fee amounts, these changes are applied during block finalization (ref↗) and take effect in the next block. Even though `finalize_gov` is called before transaction executions during `finalize_block`, checks that use parameters that can be updated through governance are not done during `finalize_block`. For example, inside `apply_wrapper_tx` `transfer_fee` is called (ref↗) without prior check if fee is above minimum fee as it is done during prepare and process proposal (ref↗).

---

## Property 03: Fee payer has sufficient funds and fee meets minimum requirements during process proposal

Let t be a transaction included in a block proposal b. A validator accepts a b in processProposal if

- the fee payer is guaranteed to have sufficient funds to cover the transaction execution, i.e., the fee payer balance of the fee token during the transaction's execution is greater than or equal to (gas limit * fee amount)
- the fee amount is greater than or equal to the minimum fee amount for the given token during the transaction's execution.

### Violation consequences

- Validators may accept blocks containing transactions with insufficient fee payer balances or inadequate fee amounts, leading to failed fee transfers during block execution. This results in transaction failures, wasted computational resources, and potential economic losses for validators who cannot collect fees from failed transactions.

### Threats

- Threat 3.1. There is no code checking that the fee payer has sufficient funds.
- Threat 3.2. There is code checking that the fee payer has sufficient funds, but it does not consider that the fee payer may also be the fee payer of some preceding transactions in the same block.
- Threat 3.3. There is no code checking that the fee amount is greater than or equal to the minimum fee amount for the given token.
- Threat 3.4. There is code checking that the fee amount is greater than or equal to the minimum fee amount for the given token but it does not consider that the minimum can be updated via governance.

#### Conclusion

The property holds.

#### Threat 3.1. conclusion

***The threat is not applicable.***

Check for sufficient funds is done in the `transfer_fee` function (ref↗). The function reads the fee payer's balance and uses `checked_sub` to verify sufficient funds before attempting the transfer. If insufficient funds are detected, MASP fee payment (ref↗) is attempted by executing the first transaction of the batch to unshield funds, then

balance check is done again. If the fee payer still has insufficient funds after the MASP payment attempt, the transaction is rejected.

#### Threat 3.2. conclusion

***The threat is not applicable.***

The temporary state mechanism in `process_proposal` (ref↗) creates an isolated environment with its own write log for transaction simulation. Using this mechanism mitigates threat because when simulating transactions, each transaction's effects, including fee payments, are accumulated in the temporary write log. All the balance changes from all preceding transactions are considered, as those changes are reflected in the write log.

#### Threat 3.3. conclusion

***The threat is not applicable.***

Minimum gas price is read only from chain parameters (ref↗) during `process_proposal`, instead of comparing consensus-mandated minimum with proposer's own minimum like in `prepare_proposal`. This is appropriate because during `process_proposal` only consensus-wide minimum validation is relevant, as local validator fee preferences are only applicable to block proposers during `prepare_proposal`. The `fee_data_check` (ref↗) ensures the fee amount per gas unit is greater than or equal to the minimum gas price for the given token.

#### Threat 3.4. conclusion

***The threat is not applicable.***

While the governance can update minimum fee amounts, these changes are applied during block finalization (ref↗) and take effect in the next block. Even though `finalize_gov` is called before transaction executions during `finalize_block`, checks that use parameters that can be updated through governance are not done during `finalize_block`. For example, inside `apply_wrapper_tx` `transfer_fee` is called (ref↗) without prior check if fee is above minimum fee as it is done during prepare and process proposal (ref↗).

---

## Property 04: An executed transfer transaction decreases the fee payer balance of the fee token by (gas limit * fee amount).

### Violation consequences

- The fee payer's balance may not be decreased by the correct amount, fee payments could be lost entirely during transaction processing, or fees might be paid multiple times for the same transaction. Users could potentially execute transactions without paying required fees, while block proposers might receive incorrect or no fee compensation.

### Threats

- Threat 4.1. There's a bug and the transfer is not executed.
- Threat 4.2. The transfer is executed but the state changes are not persisted because the transaction does not pass validation.
- Threat 4.3. The transfer is executed but the state changes are not persisted because another transaction in the same block does not pass validation.
- Threat 4.4. The transfer is executed but the state changes are not persisted because the transaction exceeds its gas limit during execution.
- Threat 4.5. The transfer is executed but the state changes are not persisted because the transaction exceeds its gas limit during validation.

- Threat 4.6. The transfer is executed but the state changes are not persisted because another transaction in the same block exceeds its gas limit during execution.
- Threat 4.7. The transfer is executed but the state changes are not persisted because another transaction in the same block exceeds the gas limit during validation.

**Conclusion**

The property holds.

**Threat 4.1. conclusion**

***The threat is not applicable.***

The fee transfer is executed as part of the wrapper transaction processing in the `apply_wrapper_tx` function. The `transfer_fee` function performs the actual token transfer from the fee payer to the block proposer using `fee_token_transfer`, which calls the `token::transfer` function that ensures the transfer is executed properly.

**Threats 4.2-4.7. conclusion**

***All threats are not applicable.***

The fee transfer is executed and committed to the block write log during wrapper transaction processing before any subsequent validations or gas checks that could fail (code ref↗), indicating that the fee payment is moved to the block write log regardless of any failures that occur later in the transaction processing pipeline. This means that validation failures, gas limit exceeded errors, or failures of other transactions in the same block cannot affect the fee payment that was already committed at the block level.

---

## Property 05: An executed transfer transaction increases the block proposer's balance of the fee token by (gas limit * fee amount)

### Violation consequences

- The block proposer's balance may not be increased by the correct amount, fee payments could be lost entirely during transaction processing. Block proposers might receive incorrect or no fee compensation.

### Threats

- Threat 5.1. There's a bug and the transfer is not executed.
- Threat 5.2. The transfer is executed but the state changes are not persisted because the transaction does not pass validation.
- Threat 5.3. The transfer is executed but the state changes are not persisted because another transaction in the same block does not pass validation.
- Threat 5.4. The transfer is executed but the state changes are not persisted because the transaction exceeds the gas limit during execution.
- Threat 5.5. The transfer is executed but the state changes are not persisted because the transaction exceeds the gas limit during validation.
- Threat 5.6. The transfer is executed but the state changes are not persisted because another transaction in the same block exceeds the gas limit during execution.
- Threat 5.7. The transfer is executed but the state changes are not persisted because another transaction in the same block exceeds the gas limit during validation.

### Conclusion

The property holds.

The analysis for Property 4 also covers the threat inspection of Property 5, since both properties describe the same fee transfer operation from different perspectives.

---

## Property 06: Let t be a transaction included in a decided block b and assume that the transaction is committed, i.e., passes validation. Then, the transaction uses at most the gas limit of the transaction.

### Violation consequences

- If a gas limit check failure is not detected, or if a transaction exceeds its gas limits and the violation is detected but its state changes are still persisted, this can result in users submitting transactions that consume more resources than they paid for. This poses both safety and liveness risks.

### Threats

- Threat 6.1. The execution of a transaction exceeds its gas limits, but the violation is not detected.
- Threat 6.2. The execution of a transaction exceeds its gas limits and the violation is detected during execution; however, its state changes are still persisted, i.e., the transaction is committed.
- Threat 6.3. The validation of a transaction exceeds its gas limits, but the violation is not detected.
- Threat 6.4. The validation of a transaction exceeds its gas limits and the violation is detected during execution; however, its state changes are still persisted, i.e., the transaction is committed

### Conclusion

The property holds.

Gas meter↗ is used to track gas consumption during transaction execution. The `consume()` (code ref↗) method accumulates gas usage and checks for overflow by comparing `transaction_gas > tx_gas_limit`, returning `TransactionGasExceededError` when exceeded.

#### Threat 6.1. and 6.2. conclusion

***Both threats are not applicable.***

The gas metering is comprehensively applied throughout the finalize block phase, ensuring no transaction execution path can bypass gas limits:

- The `retrieve_and_execute_transactions` function processes wrapper and protocol transactions by configuring dispatch arguments and creating appropriate gas meters based on transaction type, then passing them to `dispatch_tx()`. Wrapper transactions receive gas meters with transaction-specific limits (code ref↗) while protocol transactions receive zero-limit gas meters (code ref↗), making them gas-free. After successful wrapper execution, inner transactions are also executed using the same gas meter (returned via `WrapperCache`) to ensure they respect the overall transaction gas limit (code ref↗).
- Additional wrapper transactions processing gas is accounted (code ref↗). During the execution, gas is charged for functions like storage reads, writes, `fetch_or_compile()` (code ref1↗, ref2↗) for WASM module compilation, and actual WASM code execution through `apply_tx.call()`. All these operations consume gas through the passed gas meter, and when gas limits are exceeded, the gas error propagates up through the execution chain where it triggers state rollback to ensure no invalid state modifications persist (example code ref↗).

#### Threat 6.3. and 6.4. conclusion

***Both threats are not applicable.***

In the function `execute_vps`, VPs are executed in parallel (code ref↗), and each VP is assigned a `VpGasMeter` instance that tracks gas consumption independently (code ref↗). Gas meters are created from the transaction's gas meter, inheriting the transaction's gas limit and current consumption (code ref↗). Each VP only tracks its gas consumption in `current_gas` through the function `consume` (code ref↗), and the consumed gas is validated against the transaction gas limit (code ref↗).

The total gas consumed by all VPs is then checked using `merge_vp_results` (code ref↗). The total VP gas consumption is added to the transaction's gas meter, and if the combined gas exceeds limits, the entire transaction fails (code ref↗).

During the execution of MASP VP and Multitoken VP, the gas is consumed through functions `add_gas()` (code ref↗) and `charge_gas` (code ref↗), which call the function `consume` (code ref↗) from `GasMeter`.

The VPs will return an error in case the gas limit check fails (code ref↗). The gas consumed until the failure point is still counted (code ref↗). When `consume` returns an error, the `current_gas` has been updated, and `get_vp_consumed_gas` will return this accumulated amount.

---

## Property 07: Let t be a committed transfer transaction, c its corresponding transfer command and s a source component in c.sources. Then, the s.source balance for s.token is greater than or equal to the s.amount right before the execution of the transaction.

### Violation consequences

- A user could submit a transaction where the transferred amount exceeds their actual transparent or shielded balance. This would allow users to effectively mint tokens, violating the integrity of the system.

### Threats

- Threat 7.1. Assume that c is a shielding or unshielding command that includes a transfer f, i.e, f.source is a spending key. Either
  - There is no check in the MASP VP to verify that the total amount of s.token spendable by s.source right before the execution of the transaction is greater than or equal to s.amount. Note that the total amount is computed based on the notes in the note commitment tree.
  - There is a check, but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ s.amount.
- Threat 7.2. Assume that c is a shielding command that includes a transfer f, i.e., s.source is a transparent address. Either
  - There is no check to verify that s.source balance is greater than or equal to s.amount right before the execution.
  - There is a check, but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ s.amount.

### Conclusion

The property holds.

#### Threat 7.1. conclusion

The threat is not applicable in case when the `verify_shielded_tx` function is called (code ref↗), the `cv_sum` is correctly computed through `check_spend` (code ref↗), `check_convert` (code ref↗), and `check_output` (code ref↗), as the net value commitment from input and output notes. The function `final_check` adjusts `cv_sum` by `value_balance` (code ref↗). Additionally, for the threat not to be applicable, `binding_sig` needs to be validated correctly

through the `validate` method (code ref↗), which should ensure that the value balance is accurate and authorized.

Notably, functions `check_bundle` and `validate` are not part of the scope for this audit.

#### Threat 7.2. conclusion

**The threat is not applicable.**

As mentioned in the conclusion of threat 9.4, a transfer transaction applies transparent and shielded transfers via the `multi_transfer` method (code ref↗). The function `apply_transparent_transfers` (code ref↗) calls the `multi_transfer` (code ref↗), which ensures that the owner has sufficient balance for the transfer (code ref↗).

---

## Property 08: Given a transfer transaction t built via a transfer command, then it is committed if it is accepted, the transaction uses no more gas than the transaction's gas limit, and let c be the transaction's corresponding transfer command. For any source component s in c.sources, the s.source balance for s.token is greater than or equal to the s.amount right before the execution of the transaction.

### Violation consequences

- The submitted transaction can be rejected by VPs due to incorrect format of transactions generated by the builders.

### Threats

- Threat 8.1. There is a mismatch between the format of the transfer transactions generated by the builders and the format and requirements assumed by the VPs.

### Conclusion

The property holds.

#### Threat 8.1. conclusion

***The threat is not applicable.***

The builder constructs transactions using a well-defined structure, assembling all transfer components (sources, targets, fees, asset types, etc.) according to chain expectations. Transparent and shielded parts are encoded and referenced (e.g., via `shielded_section_hash`). It produces transactions that won't cause the execution of the predefined transaction to fail, as long as it is accepted, the transaction uses no more gas than the transaction's gas limit, and the source balance is greater than or equal to the amount being transferred before the execution of the transaction.

A transaction is executed only if it's an allowed transaction (code ref↗). The main entry point for transaction execution is `apply_tx`, which decodes transaction data and delegates to `token::multi_transfer`. The WASM code ( `apply_transparent_transfers`, `multi_transfer`, and `apply_shielded_transfer`) performs checks on all inputs: balances, asset types, authorizations, and value flows. These routines expect and correctly handle the structure generated by the builder, using explicit decoding and validation. Balance modifications and underflow/overflow checks are also enforced by the WASM module. If there were a format mismatch between builder and WASM expectations, transactions would fail at the deserialization or validation stage (typically with an explicit error such

as “Failed to decode …” or “insufficient balance”) and would not reach the state transition or commit phase. Gas is checked during transaction execution and validation.

---

## Property 09: Let t be a committed transfer transaction, then the balances of the involved parties decrease and increase accordingly.

More precisely:

- Let t be a committed transfer transaction,
- c its corresponding transfer command,
- s a source component in c.sources, and
- g a target componet in c.targets.

Then, the s.source balance for s.token decreases by s.amount and the g.target balance for g.token increases by g.amount.

### Violation consequences

- The user's balance can be incorrectly modified which can cause arbitrarily minting or burning of the funds. This would pose a safety issue.

### Threats

- Threat 9.1. Assume that c is a shielded transfer command such that s.source is a spending key and g.target is a payment address. Either
  - There is no check in the MASP VP to verify that new notes have been created for the spending key associated with g.target for g.token and g.amount, it can be skipped or it is
  - There exists a check but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ g.amount.
- Threat 9.2. Assume that c is a shielded transfer command such that s.source is a spending key and g.target is a payment address. Either
  - There is no check in the MASP VP to verify that enough notes are spent and that the corresponding nullifiers are revealed.
  - There exists a check but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ s.amount.
- Threat 9.3. Assume that c is a shielding transfer command such that s.source is a transparent address and g.target is a payment address. Either
  - There is no check in the MASP VP to verify that new notes have been created for the spending key associated with g.target for g.token and g.amount.
  - There exists a check in the MASP VP but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ g.amount.
- Threat 9.4. Assume that c is a shielding transfer command such that s.source is a transparent address and g.target is a payment address. Either
  - There is no check to verify the s.source balance of s.token has decreased by s.amount.
  - There exists a check but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ s.amount.
- Threat 9.5. Assume that c is an unshielding transfer command such that s.source is a spending key and g.target is a transparent address. Either
  - There is no check in the MASP VP to verify that enough notes are spent and that the corresponding nullifiers are revealed.
  - There exists a check but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ s.amount.

- Threat 9.6. Assume that c is an unshielding transfer command such that s.source is a spending key and g.target is a transparent address. Either
  - There is no check to verify the g.target balance of g.token has increased by g.amount.
  - There exists a check but it can be skipped or the check is incorrect, e.g., it checks for an amount $\neq$ g.amount.

## Conclusion

The property holds.

**Threat 9.1. conclusion**

**The threat is not applicable.**

The `valid_note_commitment_update` function validates that the note commitment tree is correctly updated with new note commitments by comparing the pre-state with the added `shielded_outputs` commitments to the post-state (code ref ↗), ensuring new notes have been created.

Additionally, the threat is not applicable in the case when the `verify_shielded_tx` function is called (code ref ↗), and the function `check_bundle` (code ref ↗) verifies the `shielded_outputs` through the function `check_output` (code ref ↗) and populates the `BatchValidator` context with the transaction's cryptographic data correctly.

Further, in order for the threat not to be applicable, the notes need to be verified through the `validate` function (code ref ↗). It should ensure `cv`, `cmu`, `ephemeral_key`, and `zkproof` of transactions `OutputDescription` are validated (code ref ↗), together with signatures (code ref ↗), validating the g.target for g.token and g.amount.

Notably, functions `check_bundle` and `validate` are not part of the scope for this audit.

**Threat 9.2. conclusion**

**The threat is not applicable.**

The function `valid_nullifiers_reveal` verifies that the nullifier hasn't already been used in a previous transaction, or if it's already been revealed within the current transaction (code ref ↗). It also ensures that the nullifiers revealed in the transaction exist in post-state storage with an empty value, confirming it has been recorded correctly without additional data (code ref ↗), and that no unneeded nullifiers have been revealed (code ref ↗).

Similarly, as noted in the conclusion of the previous threat, the function `verify_shielded_tx` verifies the proofs (code ref ↗). The threat is not applicable in case the function `check_bundle` (code ref ↗) verifies the `shielded_spends` through the function `check_spend` (code ref ↗) accurately and populates the `BatchValidator` context with the transaction's cryptographic data correctly.

Further, for the threat not to be applicable, the nullifiers are then verified through the `validate` function (code ref ↗). This function should ensure that `rk`, `cv`, `nullifier`, `anchor`, and `zkproof` of transactions `SpendDescription` are validated (code ref ↗), together with signatures (code ref ↗).

Additionally, the threat is not applicable if the `cv_sum` is correctly computed through `check_spend` (code ref ↗), `check_convert` (code ref ↗), and `check_output` (code ref ↗), as the net value commitment from input and output notes. The function `final_check` adjusts `cv_sum` by `value_balance` (code ref ↗). Also, for the threat not to be applicable, `binding_sig` needs to be validated correctly through the `validate` method (code ref ↗), which should ensure that the value balance is accurate and authorized, and enough notes are spent.

**Threat 9.3. conclusion**

**The threat is not applicable.**

Similarly, as noted in the conclusion of Threat 9.1, the `valid_note_commitment_update` function validates that the note commitment tree is correctly updated with new note commitments by comparing the pre-state with the added `shielded_outputs` commitments to the post-state (code ref ↗). Additionally, for the threat not to be

applicable, the function `verify_shielded_tx` should ensure the proofs are validated correctly (code ref ↗) through methods `check_bundle` (code ref ↗) and `validate` (code ref ↗).

#### Threat 9.4. conclusion

**The threat is not applicable.**

A transfer transaction applies transparent and shielded transfers via the `multi_transfer` method (code ref ↗). The function `apply_transparent_transfers` (code ref ↗) calls the `multi_transfer` (code ref ↗), which ensures that the owner has sufficient balance for the transfer and that the amount deducted equals the amount being received by the account, subtracted from the amount being sent from the account (code ref ↗).

The Multitoken VP ensures token changes are balanced. The `dec_changes` tracks the actual balance decreases that happened in storage (code ref ↗). If accounts lose tokens, they must either appear in other accounts or be burned (code ref ↗).

The MASP VP validates transparent inputs for shielding transfers. The `validate_transparent_bundle` function checks that the `transparent_tx_pool` balances to zero (code ref ↗), which ensures that the transparent inputs and outputs balance out with the `sapling_value_balance`.

Additionally, as mentioned in the previous conclusion of *property 10*, when `verify_sapling_balancing_value` is invoked, it ensures that the MASP balance changes match the expected balance changes from the `sapling_value_balance` (code ref ↗).

#### Threat 9.5. conclusion

**The threat is not applicable.**

Similarly, as noted in the conclusion of Threat 9.2, the function `valid_nullifiers_reveal` verifies that the nullifier hasn't already been used in a previous transaction, or if it's already been revealed within the current transaction (code ref ↗). It also ensures that the nullifiers revealed in the transaction exist in post-state storage with an empty value, confirming it has been recorded correctly without additional data (code ref ↗), and that no unneeded nullifiers have been revealed (code ref ↗). Additionally, for the threat not to be applicable, the function `verify_shielded_tx` should ensure the proofs are validated correctly (code ref ↗) through methods `check_bundle` (code ref ↗) and `validate` (code ref ↗).

#### Threat 9.6. conclusion

**The threat is not applicable.**

Similarly, as noted in the conclusion of threat 9.4, a transfer transaction applies transparent and shielded transfers via the `multi_transfer` method (code ref ↗). The function `apply_transparent_transfers` (code ref ↗) calls the `multi_transfer` (code ref ↗), which ensures that the amount increased equals the amount being sent from the account, subtracted from the amount being received by the account (code ref ↗).

The Multitoken VP ensures token changes are balanced. The `inc_changes` tracks the actual balance increases that happened in storage (code ref ↗). If accounts receive tokens, they must either come from other accounts or be minted (code ref ↗).

The MASP VP validates transparent inputs for shielding transfers. The `validate_transparent_bundle` function checks that the `transparent_tx_pool` balances to zero (code ref ↗), which ensures that the transparent inputs and outputs balance out with the `sapling_value_balance`.

Additionally, as mentioned in the previous conclusion of *property 10*, when `verify_sapling_balancing_value` is invoked, it ensures that the MASP balance changes match the expected balance changes from the `sapling_value_balance` (code ref ↗).

# Property 10: Let t be a committed shielding transaction and c its corresponding transfer command Then, for each target component g in c, the MASP balance for g.token increases by g.amount.

## Violation consequences

- If it is not verified, the amount deducted from transparent accounts could be different from what gets credited in the MASP. This would pose a safety issue.

## Threats

- Threat 10.1. There is no check in the MASP VP to verify that the MASP balance for g.token increases by g.amount.
- Threat 10.2. There exists a check, but it can be skipped or it is incorrect, e.g., it checks for an amount $\neq$ g.amount.

## Conclusion

The property holds.

The function `validate_state_and_get_transfer_data` tracks the balance changes and stores them in the `ChangedBalances` struct (code ref ↗).

When `verify_sapling_balancing_value` is invoked, it ensures that the MASP balance changes match the Sapling value balance (code ref ↗). The `validate_transparent_bundle` function ensures that the transaction is consistent with the `ChangedBalances` (code ref ↗).

Notably, the IBC-related code was not part of the audit.

### Threat 10.1. conclusion

***The threat is not applicable.***

The function `verify_sapling_balancing_value` initializes two accumulators with post-transaction values (code ref ↗) and verifies that the accumulation of balance changes from `sapling_value_balance` is equal to the calculated `ChangedBalances pre` for the MASP address (code ref ↗), as well as the total `ChangedBalances undated_pre` is equal to the accumulation of the undated balance change (code ref ↗). It guarantees that the changes match the expected balance changes.

When the `validate_transparent_bundle` function is invoked, it initializes the `transparent_tx_pool` using the transaction's `sapling_value_balance` (code ref ↗), and, if present, validates the `transparent_bundle` (code ref ↗).

When shielding, the `validate_transparent_input` method processes each input in the transparent bundle (code ref ↗). The input's value is added to the `transparent_tx_pool` (code ref ↗). If the asset type of the transaction input exists in the conversion state and its epoch matches the current MASP epoch, the input amount is subtracted from the `ChangedBalances pre` value of the `TxIn` address (code ref ↗). If the asset type is contained in the `ChangedBalances undated_tokens`, it checks that the dated counterpart of the undated asset type does not exist in the conversion tree, and subtracts the input amount from the `ChangedBalances pre` value of the `TxIn` address (code ref ↗). Otherwise, the asset is treated as unrecognized and an error is returned (code ref ↗). Additionally, the address of the `TxIn` is inserted into the `authorizers` (code ref ↗).

Similarly, the function `validate_transparent_output` processes each output in the transparent bundle (code ref ↗), as described in the conclusion of *property 11*.

The `validate_transparent_bundle` function checks that the `transparent_tx_pool` balances to zero (code ref ↗), which ensures that the transparent inputs and outputs balance out with the `sapling_value_balance`.

### Threat 10.2. conclusion

***The threat is not applicable.***

Upon shielding transaction, the `sapling_value_balance` is negative (code ref↗), and the transaction `transparent_bundle` is `Some` (code ref↗). Consequently, the functions `verify_sapling_balancing_value` and `validate_transparent_bundle` will be executed. The checked arithmetic ensures safe arithmetic operations. Additionally, the transparent transaction pool must balance to exactly zero.

---

## Property 11: Let t be a committed unshielding transaction and c its corresponding transfer command. Then, for each source s in c, the MASP balance for s.token decreases by s.amount.

### Violation consequences

- If it is not verified, the MASP balance might not decrease even though tokens were unshielded, or it might decrease by the wrong amount, which would lead to inconsistency. This would pose a safety issue.

### Threats

- Threat 11.1. There is no check in the MASP VP to verify that the MASP balance for s.token decreases by s.amount.
- Threat 11.2. There exists a check but it can be skipped or it is incorrect, e.g., it checks for an amount $\neq$ s.amount.

### Conclusion

The property holds.

As mentioned in the previous conclusion of *property 10*, when `verify_sapling_balancing_value` is invoked, it ensures that the MASP balance changes match the Sapling value balance (code ref↗). The `validate_transparent_bundle` method verifies that the transaction is consistent with the balance changes (code ref↗).

#### Threat 11.1. conclusion

***The threat is not applicable.***

When unshielding, the `validate_transparent_output` function processes each input in the transparent bundle (code ref↗). The input's value is subtracted from the `transparent_tx_pool` (code ref↗). If the asset type of the transaction input exists in the conversion state and its epoch is less than or equal to the current MASP epoch, the input amount is subtracted from the `ChangedBalances post` value of the `TxIn` address (code ref↗). If the asset type is contained in the `ChangedBalances undated_tokens`, it subtracts the input amount from the `ChangedBalances post` value of the `TxIn` address (code ref↗). Notably, there exists no check that validates that the dated counterpart of the undated asset type does not exist in the conversion tree. Otherwise asset is treated as unrecognized and an error is returned (code ref↗).

The `validate_transparent_bundle` function checks that the `transparent_tx_pool` balances to zero (code ref↗), which ensures that the transparent inputs and outputs balance out with the `sapling_value_balance`.

#### Threat 11.2. conclusion

***The threat is not applicable.***

In case of unshielding transaction, the `sapling_value_balance` is positive (code ref↗), and the transaction `transparent_bundle` is `Some` (code ref↗). Consequently, the functions `verify_sapling_balancing_value` and `validate_transparent_bundle` will be executed. The checked arithmetic ensures safe arithmetic operations. Additionally, the transparent transaction pool must balance to exactly zero.

---

## Property 12: Let t be a committed shielded transaction and c its corresponding transfer command. Then, the balance of the MASP address remains unchanged.

### Violation consequences

- If it is not verified, the MASP transparent balance could be modified which would pose a safety issue.

### Threats

- Threat 12.1. There is no check in the MASP VP to verify that the MASP balance does not change.
- Threat 12.2. There exists a check but it can be skipped or it is incorrect.

### Conclusion

The property holds.

#### Threat 12.1. conclusion

***The threat is not applicable.***

In case of a shielded transfer, the `sapling_value_balance` is zero (code ref↗), and the transaction `transparent_bundle` is `None` (code ref↗). As mentioned in the previous conclusions of *property 10* and *property 11, in* `validate_transparent_bundle`, the code verifies that the transparent transaction pool must balance to zero, which ensures that the transparent inputs and outputs balance out with the `sapling_value_balance` (code ref↗). Additionally, `verify_sapling_balancing_value` checks that the MASP balance changes match the Sapling value balance (code ref↗).

#### Threat 12.1. conclusion

***The threat is not applicable.***

During shielded transfer, the `sapling_value_balance` is zero (code ref↗), and the transaction `transparent_bundle` is `None` (code ref↗), which ensures the validation is not skipped. The checked arithmetic ensures safe arithmetic operations. Additionally, the transparent transaction pool must balance to exactly zero.

---

## Property 13: If a transaction t increases or decreases the balance associated with the MASP address, then the MASP VP is used for validation.

### Violation consequences

- If the MASP VP is not used for validation, the transaction wouldn't be properly validated. This would pose a safty issue.

### Threats

- Threat 13.1. The MASP VP is not triggered to validate a transaction that modifies the MASP address.

### Conclusion

The property holds.

A transfer transaction applies transparent and shielded transfers via the `multi_transfer` method (code ref↗).

The `balance_key` is modified through the function `apply_transparent_transfers` (code ref↗) when the `multi_transfer` function is called (code ref↗). This function collects all the accounts whose balance has been modified and writes the newly calculated changed balance into the write log (code ref↗). For each account whose balance has been increased or decreased, the value of the changed balance from the key is modified. Additionally, a verifier will be inserted into the transaction verifiers for each source and target address, and internal token address contained in the sources (code ref↗) and targets (code ref↗).

When checking the validity of a transaction through validity predicates, the function `verifiers_and_changed_keys` is called (code ref↗). This function will return the set of verifiers that were inserted into transaction verifiers during transaction execution (code ref↗) and the first address segment of storage keys that have been modified via the write log (code ref↗).

#### Threat 13.1. conclusion

***The threat is not applicable.***

For a transaction that changes the balance associated with a MASP address, the verifier set will include the multitoken address, verifiers inserted into the transaction verifier set through `apply_transparent_transfers` during execution, which includes MASP addresss if it's source or destination of transfer, and additionally the MASP address due to changed `masp_nullifier_key`, `masp_commitment_tree_key`, or `masp_undated_balance_key`, which would be modified by `apply_shielded_transfer` when the transaction is executed (code ref↗).

The function `execute_vps` will execute each verifier's validity predicate. Consequently, for a masp transfer, MASP VP (code ref↗) will be triggered. Additionally, if a transaction increases or decreases the balance associated with the MASP address, the multitoken VP will be triggered due to changed balance keys (code ref↗).

---

## Property 14: If a transaction t increases or decreases the balance associated with a given extended full viewing key, then the MASP VP is used for validation.

### Violation consequences

- If the MASP VP is not used for validation, the transaction wouldn't be properly validated. This would pose a safety issue.

### Threats

- Threat 14.1. The MASP VP is not triggered to validate a transaction that spends notes, i.e., reveals nullifiers.
- Threat 14.2. The MASP VP is not triggered to validate a transaction that creates new commitment notes.
- Threat 14.3. The MASP VP is not triggered to validate a transaction that modifies the undated balance of one or more undated tokens.

### Conclusion

The property holds.

If a transfer contains a `shielded_section_hash`, the function `apply_shielded_transfer` will be triggered (code ref↗) when the WASM code is executed. When a shielded transfer is applied, `masp_nullifier_key` (code ref↗), `masp_commitment_tree_key` (code ref↗), and `masp_undated_balance_key` will be modified (code ref↗) depending on whether the transaction spends notes, creates commitment notes, or updates the undated balance.

As mentioned in the previous conclusion of *property 13*, the verifier set will include the MASP address due to changed `masp_nullifier_key`, `masp_commitment_tree_key`, or `masp_undated_balance_key`, which will trigger the MASP VP.

We now provide conclusions for each of the threats.

**Threat 14.1. conclusion**

***The threat is not applicable.***

If a transfer contains a `shielded_section_hash`, the `apply_shielded_transfer` function will invoke `handle_masp_tx` (code ref↗), which modifies the `masp_nullifier_key` (code ref↗). This triggers the `execute_vps` function to run the MASP validity predicate.

**Threat 14.2. conclusion**

***The threat is not applicable.***

If a transfer contains a `shielded_section_hash`, the function `apply_shielded_transfer` will change the `masp_commitment_tree_key` via `update_masp_note_commitment_tree` (code ref↗), leading to the function `execute_vps` executing the masp validity predicate.

**Threat 14.3. conclusion**

***The threat is not applicable.***

The function `apply_transparent_transfers` returns the token addresses from both the source and destination. If a transfer contains a `shielded_section_hash`, the `masp_undated_balance_key` will be updated via `update_undated_balances` for all tokens (code ref↗), where the undated balance for dated tokens will remain whatever undated balance already existed for that token (code ref↗). This triggers the `execute_vps` function to run the MASP validity predicate.

---

## Property 15: The balance associated with a viewing key only decreases if the corresponding user authorizes it.

Assume that the balance associated with a viewing key decreases. We then have that the balance decreases as a consequence of the execution of a shielded or unshielding transaction t such that

- t spends some tokens held at the extended spending key associated with the viewing key, or
- t uses the extended spending key as a gas payer

### Violation consequences

If the balance associated with a viewing key decreases without the authorization of the involved user, then either (i) the protocol unintentionally burns or transfers tokens, or (ii) a malicious user exploits a vulnerability to steal or burn tokens.

### Threats

- Threat 15.1. The MASP VP accepts an unshielding or shielded transaction t that transfers some tokens from a source without its authorization.
- Threat 15.2. The protocol unintentionally spends user notes by revealing one or more nullifiers.
- Threat 15.3. The protocol unintentionally removes unspent notes from the note commitment tree.
- Threat 15.4. The protocol unintentionally modifies or deletes existing conversions.
- Threat 15.5. Transaction signatures are not verified such that a malicious user could pick up a valid committed transaction, modify it, e.g., change target to its own address, and still pass validation.

## Conclusion

***The property holds.***

Other threats may arise if the replay protection mechanism is buggy. This is out of the scope of this audit, and therefore, it has not been analyzed carefully. Furthermore, the fact that revealed nullifiers are recorded and checked during validation should prevent replay attacks concerning shielded assets.

#### Threats 15.1, 15.2, 15.3 and 15.4 conclusion

***The threats are not applicable.***

By Property 14, if a transaction changes the balance associated with a viewing key, then the MASP VP is used for validation. The MASP VP verifies via `check_spend` that the user has the authorization to spend the involved notes and the `valid_nullifiers_reveal` makes sure that the revealed nullifiers are recorded appropriately to prevent replay attacks. Thus ***threat 15.1. is not applicable.***

We now argue that the balance associated with a viewing key can only decrease via a transaction, which would yield threats 15.2, 15.3, and 15.4 not applicable. The balance associated with a viewing is the sum of all unspent notes (those in the commitment tree whose nullifier has not been revealed) + conversions (for a given ***dated*** unspent note, the rewards from the epoch the note was created until the current epoch). Let S be the set of unspent notes owned by the corresponding spending key. To reduce the balance associated with the viewing key outside the context of a valid transaction, the protocol must take any of the following actions:

1. Spend some notes in S by revealing one or more nullifiers
2. Modify the note commitment tree in a way that some notes in S do not belong to the tree after the modification
3. Modify an existing conversion with epoch e in E, where E is the set of MASP epochs in the notes in S.

We now analyze each case.

Case 1. A nullifier is revealed via the `reveal_nullifiers` function in `crates/shielded_token/src/utils.rs`. The function `handle_masp_tx` is its only reference, which is only called by the `apply_shielded_transfer` function when executing a transfer transaction. Thus, case 1 is impossible and ***threat 15.2 is not applicable.***

Case 2. The note commitment tree can be updated via the `update_note_commitment_tree` function in `crates/shielded_token/src/utils.rs`. This function is only called through `apply_shielded_transfer` when executing a transfer transaction. Thus, case 2 is only possible if the `update_note_commitment_tree` updates the tree wrongly by removing existing notes or modifying their descriptions. Nevertheless, the function simply appends notes to the existing tree. Thus, case 2 is impossible as well and ***threat 15.3 is not applicable.***

Case 3. Existing conversions are update in `update_allowed_conversions` at the end of a MASP epoch. Let e be the new MASP epoch. The function only adds the latest conversion (to epoch e) without modifying the already computed ones. Thus, case 3 is also impossible and ***threat 15.4 is not applicable.***

This has been inspected in detail in a previous audit. In this audit, we have audited the changes to the involved code, which are minor, between the commit of the previous audit (b24938efd948cb43fc996a729138cb099abcadc0) and this audit.

#### Threats 15.5 conclusion

***The threat is not applicable***

Transactions are signed and signatures are verified during execution. If a dishonest user modifies the internals of a transaction, then the signature verification will fail.

---

## Property 16: The balance associated with a viewing key only increases as a consequence of a shielding or shielded transaction that authorizes a transfer targeting the viewing key.

Assume that the balance associated with a viewing key increases. We then have that the balance increases as a consequence of the execution of a shielded or shielding that results in the creation of notes owned by the extended spending key associated with the viewing key.

Note that the above property does not hold for native tokens as the balance of any viewing key holding dated assets in MASP will increase at the end of a MASP epoch due to conversions. We disregard this case in our analysis.

### Violation consequences

- If the balance associated with a viewing key increases without the authorization of the involved user, then either (i) the protocol unintentionally mints or transfers tokens, or (ii) a malicious user exploits a vulnerability to mint tokens.

### Threats

- Threat 16.1. The protocol unintentionally creates user notes.
- Threat 16.2. The protocol unintentionally rolls back the revealing of nullifiers.
- Threat 16.3. The protocol unintentionally modifies existing conversions, or creates new ones between MASP epochs.
- Threat 16.4. Transaction signatures are not verified such that a malicious user could pick up a valid committed transaction, modify it, e.g., change the target to its own address, and still pass validation.

### Conclusion

***The property holds.***

Other threats may arise if the replay protection mechanism is buggy. This is out of the scope of this audit, and therefore, it has not been carefully analyzed.

#### Threats 16.1, 16.2, and 16.3 conclusion

***The threats are not applicable.***

By Property 14, if a transaction changes the balance associated with a viewing key, then the MASP VP is used for validation.

We now argue that the balance associated with a viewing key can only increase via a transaction, which would yield threats 16.1, 16.2, and 16.3 not applicable. The balance associated with a viewing is the sum of all unspent notes (those in the commitment tree whose nullifier has not been revealed) + conversions (for a given ***dated*** unspent note, the rewards from the epoch the note was created until the current epoch). Let S be the set of unspent notes owned by the corresponding spending key. To increase the balance associated with the viewing key outside the context of a valid transaction, the protocol must take any of the following actions:

1. Add new notes to S
2. Forget that some nullifiers associated with notes in S had been revealed
3. Add conversions between MASP epochs, or modify existing ones at any point

We now analyze each case.

Case 1. The revelation of a nullifier could only be rolled back by deleting it from storage (via the `delete` function of the `StorageWrite` trait). There is no code doing so. Thus, case 1 is impossible, and ***threat 16.1 is not applicable.***

Case 2. The note commitment tree can be updated via the `update_note_commitment_tree` function in `crates/shielded_token/src/utils.rs`. This function is only called through `apply_shielded_transfer` when

executing a transfer transaction. Thus, case 2 is only possible if the `update_note_commitment_tree` updates the tree wrongly by adding notes associated with spending keys that are not involved in the transfer or modifying the descriptions of existing ones. This is not the case: the `update_note_commitment_tree` simply appends notes from the transfer transaction's `bundle.shielded_outputs`. Thus, case 2 is impossible, and ***threat 16.2 is not applicable.***

Case 3. Existing conversions are only updated in `update_allowed_conversions` at the end of a MASP epoch. Thus, case 3 is only possible if the protocol modifies existing ones at the end of a given MASP epoch e-1. The function only adds the latest conversion (to epoch e) without modifying the already computed ones. Thus, case 3 is also impossible and ***threat 16.3 is not applicable.***

This has been inspected in detail in a previous audit. In this audit, we have audited the changes to the involved code, which are minor, between the commit of the previous audit (b24938efd948cb43fc996a729138cb099abcadc0) and this audit.

**Threats 16.5 conclusion**

***The threat is not applicable***

Transactions are signed and signatures are verified during execution. If a dishonest user modifies the internals of a transaction, then the signature verification will fail.

---

## Property 17: Let n be an honest node. After a client synchronizes its shielded state with n via shielded sync the client's note commitment tree matches the node's note commitment tree, the client's the notes index matches the node's the notes index, the client's the witnesses map matches the node's the witnesses map, the client's the set of nullifiers matches the node's the set of nullifiers, the set of non-spent notes owned by the client's spending key

### Violation consequences

Violations of Property 17 can result in a range of security and usability issues, from transaction failures to loss of funds or privacy:

- If there is a bug in the synchronization of the note commitment tree or witnesses map (Threats 17.2, 17.3, 17.4), the client may attempt to spend notes with invalid or outdated witnesses. This will generally result in transaction rejection by the validator, compromising liveness.
- If the notes index is not properly synchronized, the wallet may display incorrect balances, omit spendable notes, or allow double use of notes, potentially causing the client to attempt invalid transactions or miss available funds.
- If the client's set of nullifiers is out of sync with the node's state (Threat 17.1), spent notes may be displayed as spendable, enabling the user to attempt double spends that will always fail validation, or, hiding valid spendable notes from the user and causing accidental fund loss.
- If any of the above structures is only partially synchronized, the wallet state may become inconsistent, leading to transaction malleability, lost change, incorrect balances, or in the worst case, the client constructing transactions that cannot be validated or included on-chain.

## Threats

- Threat 17.1. The computation of nullifiers is incorrect, e.g., the client believes that some revealed nullifiers have not yet been revealed or vice versa.
- Threat 17.2. The commitment notes tree fetcher is buggy and does not return the correct state from the indexer.
- Threat 17.3. The commitment notes tree is buggy and does not fetch the correct state.
- Threat 17.4. The witnesses map fetcher is buggy and does not fetch the correct state.
- Threat 17.5. Fetched MASP transactions are processed out of order.
- Threat 17.6. The code does not deal correctly with fetching MASP state at different height from an indexer, e.g., in a single execution of the dispatcher, the note commitment tree is fetched at a height h and the witnesses map at a height > h.
- Threat 17.7. Applying deltas to MASP state is done incorrectly.

## Conclusion

***The property holds.***

However, two low severity findings were uncovered during the inspection of the threats related to this property.

- **Threat 17.1. Conclusion: The Threat is not applicable.** Nullifier computation and tracking are handled in the functions `save_shielded_spends` and `save_decrypted_shielded_outputs`, both called from `apply_cache_to_shielded_context` during state application. Each shielded spend's nullifier is checked against the wallet's `nf_map`, and spent notes are marked only if the corresponding nullifier is present. Nullifier updates are applied in deterministic transaction/block order as enforced by the iteration over `self.cache.fetched.take()`. There are no code paths that can skip, double-count, or roll back nullifiers, because all updates occur atomically and strictly in batch/block order when state is committed. No partial or duplicate nullifier changes are possible.

- **Threat 17.2. Conclusion: The threat is not applicable.** The commitment tree is fetched at the latest synced block height by the `fetch_commitment_tree` function. This function attempts to fully deserialize the tree using `BorshDeserialize::try_from_slice(&payload.commitment_tree)` and only returns a value on complete success; if deserialization fails, an error is returned and the cache remains unset. The only point at which the wallet context's tree is updated is in `apply_cache_to_shielded_context`, with this code:

```
if let Some((_, cmt)) = self.cache.commitment_tree.take() {
self.ctx.tree = cmt;
}
```

  Thus, the in-memory commitment tree is only replaced if a fully valid tree exists in the cache, and invalid or partial trees are never committed. Any fetch or decode error results in a retry or explicit failure, never a partial update. When using the indexer, the shielded sync process fetches the commitment tree only at the latest queried block height. The indexer's API is designed to return the full state of the tree as of that height, so a single fetch is sufficient to obtain a consistent and complete view. This conclusion assumes nullifier computation is correct (see Threat 17.1).

- **Threat 17.3. Conclusion: The threat is not applicable.** The application of the commitment tree to the wallet context is performed in `apply_cache_to_shielded_context`, which updates the commitment tree only if a valid cache entry exists (see code in threat 17.2). The commitment tree cannot be updated unless a complete, valid tree is present for the latest synced height. Block ordering and cache atomicity are enforced by the structure of `apply_cache_to_shielded_context` and its iteration over fetched transaction batches. As a result, incomplete caches, unordered application, or out-of-order/partial updates are prevented by construction. If the cache does not have a valid tree, the context is left unmodified. This conclusion assumes that the commitment tree fetcher is correct (see Threat 17.2). This conclusion assumes nullifier computation is correct (see Threat 17.1).

- **Threat 17.4. Conclusion: The threat is not applicable.** The witness map is fetched using `fetch_witness_map` at the latest queried height. The result is only stored in the cache if all witness entries are successfully deserialized;

if any error occurs, the cache remains unchanged and the operation is retried. The witness map is applied to the wallet context in `apply_cache_to_shielded_context` with:

```
if let Some((_, wm)) = self.cache.witness_map.take() {
    self.ctx.witness_map = wm;
}
```

This guarantees that only a fully valid, successfully fetched witness map is ever committed to the wallet context. If the cache is incomplete or contains errors, the context is not updated. The witness map is also fetched only at the latest queried block height in indexer mode. The indexer is designed to provide the best available witness map for the requested height, but may, in some scenarios, return the map for the closest available height less than or equal to the requested one. The shielded sync code accepts this response and applies the witness map as the state at the query height. While this is generally safe (as witnesses only become valid after their creation and should remain valid for subsequent heights), if the closest available witness map is for a height strictly less than the requested height, there is a theoretical risk of the client missing a newly added witness. In practice, this is mitigated by the fact that the sync process is always moving forward and witnesses are only required for notes up to the last synced block. This conclusion assumes nullifier computation is correct (see Threat 17.1).

- **Threat 17.5. Conclusion: The threat is not applicable.** Fetched MASP transactions are always processed in block order due to the design of the cache and fetch routines. The `Fetched` cache only accepts new transactions through the `extend` method, which takes sorted input and is implemented using a `BTreeMap`, maintaining order by key (`Fetched::extend, Fetched::iter`). In the dispatcher's `handle_incoming_message`, batches are only added after all transactions in the range have been successfully fetched and deserialized. The helper function `blocks_left_to_fetch` ensures that missing blocks are detected and retried before any extension occurs. No code path allows for out-of-order insertion or partial batch updates; transactions for each block are processed as atomic batches and always in ascending block height order.

- **Threat 17.6. Conclusion: The threat is not applicable.** The dispatcher spawns separate tasks for updating the note index, commitment tree, and witness map, each at an explicit height (`spawn_update_note_index`, `spawn_update_commitment_tree`, `spawn_update_witness_map`). In `apply_cache_to_shielded_context`, only one instance of each state element (note index, commitment tree, witness map) is applied per sync, and only if it matches the requested (latest) height; mismatched or partial state is never committed. The client always requests each of these state components at the same (latest) block height, as confirmed in the sync task logic. The code does not attempt to stitch together state from different heights within the same sync. If the indexer returns a witness map for a height less than the requested one, the code applies it as the best available map for the queried height. While this may result in a missing witness for the very latest note, the sync process corrects this in the next run, and no inconsistency or corruption can result.

- **Threat 17.7. Conclusion: The threat is not applicable.** MASP state updates such as adding shielded spends, updating witness maps, or saving decrypted notes are applied in strict block and transaction order as established by the `Fetched` cache and the iteration in `apply_cache_to_shielded_context`. Updates are performed in a single pass after all required data has been fetched and deserialized, and only after ordering invariants are established. The relevant functions, `save_shielded_spends` and `save_decrypted_shielded_outputs`, operate only after batch and index consistency has been checked. No code path exists that could apply deltas multiple times, miss updates, or introduce state divergence. The cache application is atomic, and context updates only occur after successful and complete processing, so restarts, retries, or rollbacks do not break these invariants. This conclusion assumes that the ordering invariant of fetched transactions (see Threat 17.5) holds.

---

## Property 18: Let c be a transfer command. Assume that the command is successfully executed and let t be the resulting transaction. The transaction is then well-formed.

That the transaction is well-formed implies the following:

- Let s be a source component in c.sources and assume that s.source is a spending key. Then, the set of notes in t's shielded spends owned by s.source plus the set of conversions sum a total amount greater than or equal to s.amount.
- Let s be a source component in c.sources and assume that s.source is a spending key. Assume that the set of notes in t's shielded spends owned by s.source plus the set of conversions sum a total amount (total) greater than s.amount. Then, there is a shielded output with the change such that (total - change = s.amount)
- Let g be a target component in c.targets and assume that g.target is a shielded payment address. Then, the set of shielded outputs owned by g.target sums up to an amount equal to g.amount.
- Let s be a source component in c.sources and assume that s.source is a Namada address. Then, there is a transparent input in t for s.amount by s.source.
- Let g be a target component in c.targets and assume that g.target is a Namada address. Then, there is a transparent output in t for g.amount addressed to g.target.
- c.fee_payer == t.fee_payer
- c.fee_token == t.fee_token
- c.fee_amount == t.fee_amount
- c.gas_limit == t.gas_limit

### Violation consequences

There might be multiple implications. Many of them would cause the transaction to fail validation, i.e., compromise liveness:

- If there is a bug in the computation of conversions (threat 18.3), the transaction may fail validation because a source may not have enough funds to make the transfer.
- If there is a bug in the change computation (threat 18.9), the MASP VP may reject the transaction because some assets would be burnt, which should be disallowed.

Other issues may not cause the transaction to fail validation but could make users to pay extra fees or transfer amounts that were not intended. For instance, if there is a bug in the sources/target folding mechanism (threat 18.1).

### Threats

- Threat 18.1. There is a bug in the sources/target folding mechanism and the final amounts are incorrect.
- Threat 18.2. There is a bug in the computation of shielded spends of a given source.
- Threat 18.3. There is a bug in the computation of shielded outputs of a given target.
- Threat 18.4. There is a bug in the computation of conversions for a given source.
- Threat 18.5. There is a bug in the computation of transparent inputs of a given source.
- Threat 18.6. There is a bug in the computation of transparent outputs of a given target.
- Threat 18.7. There is a bug in the computation of nullifiers.
- Threat 18.8. The transaction's expiration height is computed incorrectly
- Threat 18.9. The transactions fee-related data does not correspond to the one submitted by the user, e.g., the fee amount is wrong.
- Threat 18.10. Change computation is buggy.
- Threat 18.11. There is a bug in the computation of MASP asset from tokens.

## Conclusion

***The property does not hold.***

#### Threat 18.1. conclusion

***The threat is not applicable.***

The folding occurs in `combine_data_for_masp_transfer` and it simply groups all tokens for a given source/target, as well as extracts all denoms. If the fee is meant to be paid via MASP, then it adds it to the corresponding source and target.

#### Threat 18.2. conclusion

***The threat is not applicable.***

The selection of shielded spends occurs in `collect_unspent_notes`↗ as part of the `add_inputs` routine. This function iterates over all unspent notes for the source's spending key and accumulates them until the required amount is reached. Double-spend protection is enforced in two ways: by checking that each note is not already marked as spent in `self.spents` (previous transactions), and by ensuring that a note is not reused within the same transaction using `spent_notes`.

Asset values are accumulated and, if necessary, converted using `compute_exchanged_amount`. The note selection process halts as soon as the sum covers the required amount. Notes are only included if they have corresponding entries in the wallet's `note_map`, `witness_map`, and `div_map`. If any required data is missing or conversion fails, the transaction build aborts with an error.

Finally, all selected notes are added to the transaction using `add_sapling_spend`. Overflow and negative value checks are present throughout, and notes not convertible to the target asset type are safely excluded.

#### Threat 18.3. conclusion

***The threat is not applicable.***

Shielded outputs are constructed in `add_outputs`, where for each target the function calculates the required amount and iterates over the value balance to create outputs using `add_sapling_output`. The code ensures a valid recipient is specified (either a shielded or transparent address), returning an error otherwise. The output value is set to the minimum of what is available and what is required (`min(rem_amount, val)`), which prevents both overpayment and underpayment. Only non-negative values are processed, and if the required amount cannot be satisfied exactly, the function returns an `InsufficientFunds` error. All output amounts are further limited by `MAX_MONEY` in `add_sapling_output`, ensuring proper value boundaries and preventing overflow.

#### Threat 18.4. conclusion

***The threat is not applicable.***

The computation of conversions is handled in `compute_exchanged_amount` and `apply_conversion`. The logic attempts to convert as much of the input asset as possible into the required asset type, using `checked` arithmetic to avoid overflows or underflows. Any remaining (non-convertible) amount - the trace is carried forward and added as-is. Only assets that contribute toward the target are selected, as determined by `is_amount_required`. For every successful conversion, a conversion note is recorded via `add_sapling_convert`. If no conversion is possible for an asset, the value is simply included in the output without alteration. This design ensures that all contributions are explicit, overflows are prevented, and the function fails safely in cases of invalid or insufficient conversions.

#### Threat 18.5. conclusion

***The threat is not applicable.***

The computation of transparent inputs is handled within the transparent input loop in `add_inputs`, where each asset and denomination position is denominated using `digit.denominate(amount)`, and nonzero amounts are added as transparent inputs via `add_transparent_input`. The code explicitly checks for zero values before adding inputs. All operations on values are performed with integer types, and there are no arithmetic operations or type casts in this path that could result in overflow. Both `add_input` and `add_transparent_input` propagate errors, aborting transaction construction in the case of a failure. Transparent inputs are constructed and pushed sequentially, ensuring no double-spend of the same coin within a transaction.

#### Threat 18.6 conclusion

***The threat is not applicable.***

Transparent outputs are constructed using `add_transparent_output`, which internally invokes `.add_output` to append the output to the transaction's `vout`. The value for each output is cast to `u64` and strictly bounded by `MAX_MONEY`, preventing any overflow or invalid value from being accepted. No unchecked arithmetic is present, and any error encountered in this process is immediately propagated as a build failure. Asset type and denomination are enforced per output, ensuring outputs are correctly formed. This design prevents errors or inconsistencies in the computation of transparent outputs for a given target.

#### Threat 18.7 conclusion

***The threat is not applicable.***

Nullifiers are computed during transaction construction starting from `build_shielded_transfer` and finally in `sapling_builder.build`. Each shielded spend input is registered using `builder.add_sapling_spend`, and the nullifier is derived via `note.nf` using the note commitment, its position in the tree, and the nullifier deriving key. This derivation, based on BLAKE2s hashing and unique parameters, guarantees that two distinct notes cannot produce the same nullifier, ensuring double-spend protection. The code uses `jubjub::Fr::from(position)` to encode the note position, which is safe for all u64 values. Errors or missing data in the computation path abort transaction building.

#### Threat 18.8 conclusion

***The threat is applicable.***

The expiration height computation occurs in `gen_shielded_transfer`:

- If the user does not provide an expiration date, then the code assigns it `u32::MAX - 20` to mimic a never-expiring transaction.
- If the user provides an expiration time, then the code computes the expiration height by adding a number of heights to the current one. The added number is computed based on the difference between the current time, the expiration time provided by the user, and an estimate of the block time.

There are two potential problems with the current code:

- The builder adds 20 extra blocks to the computed expiration height. In the case when the user provides an expiration time, there is no check that the computed expiration height is at most `u32::MAX - 20`.
- The following code may overflow:

```
u32::try_from(last_block_height)
                    .map_err(|e| TransferErr::General(e.to_string()))?
                    + delta_blocks
```

#### Threat 18.9 conclusion

***The threat is applicable.***

The builders validate fee-related data in the `validate_fee` function. Let `minimum_fee` be the minimum amount retrieved from the node and `fee_amount` the one provided by the user as input. If the user does not set the force flag to true (the relevant case), the function returns either an error, e.g., if there is no minimum fee for that token in the fetched state, or the maximum between `minimum_fee` and `fee_amount` . When `fee_amount` is less than `minimum_amount`, then the transaction will use a fee amount that it is greater than the one provided by the user. This shows that the threat is applicable. Nevertheless, rather than an issue, we consider this a valid design choice.

The builders guarantee that the other fee-related metadata is consistent with the user input.

#### Threat 18.10 conclusion

***The threat is not applicable.***

Change computation is handled in the `compute_change` function within `add_inputs`, which calculates the excess amount after covering the required output and returns it to the sender per asset type and denomination. All assets and denominations are validated for correct digit positions and values; overflow checks are enforced, and negative values are rejected. If the sum of selected inputs does not fully cover the outputs, transaction construction fails with an insufficient funds error, and no change is returned. When creating change outputs, values are limited by `MAX_MONEY`. This prevents overflows and enforces protocol limits. The design ensures that either all assets are properly returned or the transaction aborts safely, and errors in change computation are surfaced to the user.

#### Threat 18.11 conclusion

***The threat is not applicable.***

The construction of MASP asset types from tokens is performed in `construct_shielded_parts` by invoking `precompute_asset_types`, which generates all possible asset types for each token and `MaspDigitPos` via `encode_asset_type`. The resulting `AssetType` includes the token, denomination, and digit position. During note selection in `collect_unspent_notes`, only notes with asset types exactly matching the required digit position and denomination are considered for spending. There is no code path that allows a note from one digit position to be used to fulfill an amount for a different position. If the transfer requests an amount in a digit position for which there are no corresponding notes, construction fails with an error. All casts from `u128` to `u64` are performed via checked conversions, and errors are propagated accordingly. As such, the implementation enforces strict asset type matching and safe casting, preventing the threat described.

---

# Findings

| Finding | Type | Severity | Status |
|---|---|---|---|
| Transactions doing masp fee payment may be executed an unbounded number of times for free | Implementation | Medium | Acknowledged |
| Inconsistent handling of overflowing transactions between prepare and process proposal | Implementation | Medium | Acknowledged |
| Overflow due to expiration height computation | Implementation | Low | Acknowledged |
| Potential height inconsistency between shielded state components when fetching from indexers | Implementation | Low | Acknowledged |
| Shielded sync recovery after dishonest node | Implementation | Informational | Acknowledged |
| Unused recoverable error logic in process proposal and finalize block | Implementation | Informational | Acknowledged |

# Transactions doing masp fee payment may be executed an unbounded number of times for free

**Severity** Medium **Impact** 3 - High **Exploitability** 1 - Low

**Type** Implementation **Status** Acknowledged

## Involved artifacts

- `crates/node/src/shell/mod.rs`↗

## Description

Transactions are allowed to pay fees via masp. To do so, they must either include a transaction in the batch that unshields enough tokens to the fee payer transparent address or unshield the tokens themselves in the case of an unshielding or fully shielded transaction.

Namada currently executes such unshielding transactions in checktx both when a validator receives a transaction via gossip, e.g., before including it in its mempool, and on mempool rechecks, i.e., after a block is committed. At this point, Namada is not certain that fees can be collected from such transactions.

Due to mempool rechecks, this type of transactions may be executed an unbonded number of times and still never be included in a block, which may result in wasting a lot of computation.

## Problem scenarios

Let `t` be an unshielding transaction that does masp fee payment. Assume that `t` enters the mempool at all validators. The following scenario could repeat an unbounded number of times:

- The leader of the next (height, round) does not include `t` in its proposal
- The proposal gets accepted and committed
- `t` is successfully executed via mempool rechecks, i.e., the transaction stays in the mempool.

It is likely that `t` is **eventually** included in a proposal. There are two possibilities here:

- Best case scenario: the user still have enough funds to pay for the fees at the current height. The transaction would have been executed for free an unbounded number of times but at least some fees are paid.
- Worst case scenario: the user does not have enough funds to pay for the fees at the current height. As a result, the transaction has been executed an unbounded number of times, and no fees can be collected.

## Recommendation

We recommend avoiding the execution of transactions doing masp fee payment on mempool rechecks to bound the number of times that Namada executes them for free to at most three (first checktx, prepare and process proposal).

A good compromise could be to execute them in checktx before entering the mempool for a quick filter (if accepted it is likely that you will be able to collect fees), and avoid its execution during the recheck. To achieve this, one could maintain a map at the application to keep track of the hashes of transactions that failed WASM execution. This map could be filled up during prepare and process proposal: if a validator executes a transaction and fails during the WASM execution, then it should include it in the map. Then, during mempool recheck, the application could check this map and remove it efficiently, i.e., without executing it, the already failed transactions. It is important that the map is not part of the application state, i.e., it does not contribute to the Apphash, as each validator may have a different set of failed transactions at a given time.

# Inconsistent handling of overflowing transactions between prepare and process proposal

**Severity** Medium **Impact** 1 - Low **Exploitability** 3 - High

**Type** Implementation **Status** Acknowledged

## Involved artifacts

- `crates/node/src/shell/prepare_proposal.rs`↗
- `crates/node/src/shell/process_proposal.rs`↗
- `crates/tx/src/data/mod.rs`↗
- `crates/node/src/shell/block_alloc.rs`↗

## Description

The `AllocFailure::OverflowsBin` occurs when a transaction's size exceeds the maximum configured block space (ref↗).

There is an inconsistency in how `AllocFailure::OverflowsBin` errors are handled between the `prepare_proposal` and `process_proposal` phases of block validation. In `prepare_proposal`, transactions that cause `AllocFailure::OverflowsBin` are included in the block proposal (with a warning log, no space is actually allocated for them) (ref↗), while in `process_proposal`, the same error type is classified as non-recoverable (ref↗) and causes the entire block to be rejected (ref↗).

This code flow is used as a temporary workaround to flush large transactions that would overflow (hence never be accepted during consensus) block from the mempool.

## Problem scenarios

- A validator includes a transaction that causes `AllocFailure::OverflowsBin` in their block proposal during `prepare_proposal`. When other validators process this proposal in `process_proposal` they will reject the entire block due to the non-recoverable `AllocationError`.
- The current approach of including overflowing transactions in proposals as a mempool flushing mechanism is not ideal and creates validation inconsistencies.

## Recommendation

Add overflow validation checks in the mempool (`CheckTx`) to prevent overflowing transactions from entering the mempool in the first place, eliminating the need for the current workaround.

## Overflow due to expiration height computation

**Severity** Low **Impact** 1 - Low **Exploitability** 1 - Low

**Type** Implementation **Status** Acknowledged

### Involved artifacts

- `crates/shielded_token/src/masp/shielded_wallet.rs`↗

### Description

The expiration height computation occurs in `gen_shielded_transfer`:

- If the user does not provide an expiration date, then the code assigns it `u32::MAX - 20` to mimic a never-expiring transaction.
- If the user provides an expiration time, then the code computes the expiration height by adding several heights to the current one. The added number is computed based on the difference between the current time, the expiration time provided by the user, and an estimate of the block time.

### Problem scenarios

There are two potential problems with the current code:

- The builder adds 20 extra blocks to the computed expiration height. In the case when the user provides an expiration time, there is no check that the computed expiration height is at most `u32::MAX - 20`.
- The following code may overflow:

```
u32::try_from(last_block_height)
     .map_err(|e| TransferErr::General(e.to_string()))?
     + delta_blocks
```

### Recommendation

Add checks to ensure that the computed expiration height is at most `u32::MAX - 20` in every case and return an error otherwise.

# Potential height inconsistency between shielded state components when fetching from indexers

**Severity** Low **Impact** 2 - Medium **Exploitability** 1 - Low

**Type** Implementation **Status** Acknowledged

## Involved artifacts

- `crates/apps_lib/src/client/masp.rs`↗

## Description

The indexer is queried by the shielded sync process for shielded state structures (witness map, commitment tree, and notes indices) at a specific `latest_query_height`. However, the current indexer implementation returns the data for the closest available height that it is less than or equal to the requested one, rather than requiring an exact match.

## Problem scenarios

Correct operation requires that all shielded state components originate from the same block height: user transactions will likely be rejected otherwise. Although unlikely, the current design may result in shielded components being fetched at different heights if for instance:

- The indexer operator rolls back the indexer state to a previous height in between queries.
- Different CDN caches are used to serve the queries.

## Recommendation

Indexer and client should support an explicit "exact match" flag for shielded state queries, ensuring that all returned structures correspond precisely to the requested block height, or return an error if the requested height isn't available.

## Shielded sync recovery after dishonest node

**Severity** Informational **Impact** 0 - None **Exploitability** 0 - None

**Type** Implementation **Status** Acknowledged

### Involved artifacts

- `crates/apps_lib/src/client/masp.rs`↗

### Description

During shielded sync, all fetched transactions are accumulated in `self.cache.fetched` and then applied to the wallet context using `apply_cache_to_shielded_context`. This process mutates the wallet's internal state (`self.ctx`), which is then saved with `self.ctx.save().await`, marking the wallet as "synced" up to the last block.

### Problem scenarios

If a client initially syncs via a misbehaving node, invalid or malicious transactions may be stored in the local wallet state. Upon reconnecting to an honest node, the wallet only fetches and applies new transactions from the next unsynced block height, never re-downloading or validating historical blocks unless the context is manually purged. As a result, any state corruption from the initial dishonest sync persists indefinitely.

Users are left to diagnose the issue only after observing failed transaction attempts or rejected spends, with the only mitigation being to manually delete their wallet state and re-sync from block height 1.

### Recommendation

We recommend to document this and similar limitations for users, as well as define common practices such that users are able to recover from these situations easily.

## Unused recoverable error logic in process proposal and finalize block

**Severity** Informational **Impact** 0 - None **Exploitability** 0 - None

**Type** Implementation **Status** Acknowledged

### Involved artifacts

- crates/node/src/shell/process_proposal.rs↗
- crates/tx/src/data/mod.rs↗
- crates/node/src/shell/finalize_block.rs↗

### Description

The codebase contains unused logic in `process_proposal` and `finalize_block` related to the `is_recoverable` error classification. This logic was designed to handle `WasmRuntimeError` as a the only recoverable error (code ref1↗, ref2↗).

Nevertheless, `WasmRuntimeError` cannot be returned in any way in `process_proposal` because all errors produced during fee check are converted into `ResultCode::FeeError` (code ref↗), which is not recoverable. Therefore the logic to handle recoverable errors in `process_proposal` is unnecessary. The same applies for the logic in `finalize_block` .

### Problem scenarios

Dead code makes the codebase confusing and error prone.

### Recommendation

Remove all the recoverable/non-recoverable error logic from `process_proposal` and `finalize_block`.

# Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1↗, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score↗, and the Exploitability score↗. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale↗, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

## Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

| ImpactScore | Examples |
|---|---|
| High | Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic |
| Medium | Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x) |
| Low | Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction) |

| ImpactScore | Examples |
|---|---|
| None | Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation |

## Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

| ExploitabilityScore | Examples |
|---|---|
| High | illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors |
| Medium | illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors |
| Low | illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors |
| None | illegitimate actions taken in a coordinated fashion by all actors |

## Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

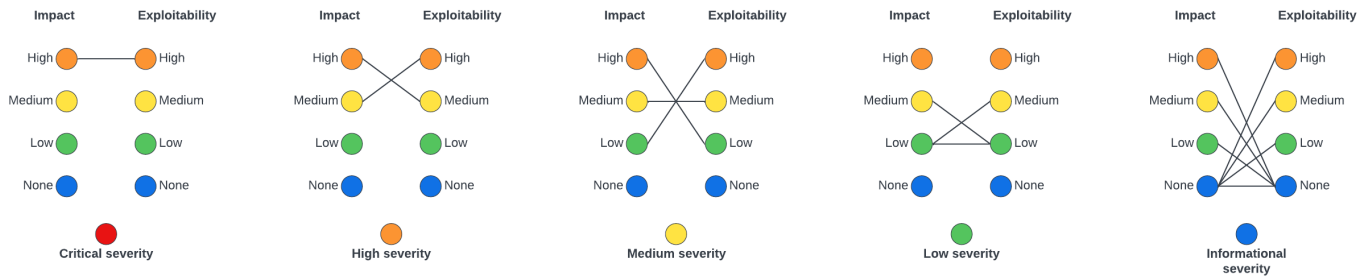


Figure 1: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

| SeverityScore | Examples |
|---|---|
| Critical | Halting of chain via a submission of a specially crafted transaction |
| High | Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers |
| Medium | Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users |
| Low | 2x increase in node computational requirements via coordinated withdrawal of all user tokens |
| Informational | Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary |

# Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.