Security Audit Report

Namada ABCI, Replay Protection, Fee And Gas Metering

Authors: Manuel Bravo, Aleksandar Ljahovic, Ivan Golubovic

Last revised 30 April, 2024

Table of Contents

Audit Overview

Scope

In April 2024, Informal Systems conducted a security audit for Heliax. The audit aimed at inspecting the correctness and security properties of Namada's ABCI custom implementation, fee system and gas metering and replay protection implementation, with a focus on the following components:

- The following files in https://github.com/anoma/namada
    - crates/apps/src/lib/node/ledger/shims
    - crates/apps/src/lib/node/ledger/shell
    - crates/apps/src/lib/node/ledger/mod.rs
    - crates/apps/src/lib/node/ledger/storage/rocksdb.rs (partially)
    - crates/apps/src/lib/node/ledger/shell/mod.rs (partially)
    - crates/gas
    - crates/state/src/wl_state.rs (partially)
    - crates/state/src/write_log.rs (partially)
    - crates/namada/src/ledger/protocol/mod.rs (partially)
    - crates/replay_protection/src/lib.rs
    - crates/parameters/src

The audit was performed from January 18, 2024 to February 22, 2024 by the following personnel:

- Manuel Bravo
- Ivan Golubovic
- Aleksandar Ljahovic

Relevant Code Commits

The audited code was from the repository at the following commit:

- `anoma/namada` : hash `f4c838d2c53b28091af9b9a2f10b5b53e55fe65e`

Conclusion

We performed a thorough review of the project. We found some subtle problems - more on them in the section Findings. Those problems, if left unattended, would violate liveness properties.

We are glad to report that the dev team has acknowledged our findings and is working towards fixing them.

Audit Dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: Rust
- **Artifacts**: https://github.com/anoma/namada

Engagement Summary

- **Dates**: 28.03.2024 - 30.04.2024
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	**#**
Critical	2
High	0
Medium	1
Low	0
Informational	5
Total	**8**

System Overview

Namada is a proof-of-stake layer-one blockchain, powered by the CometBFT (formerly Tendermint) BFT consensus algorithm. This blockchain ecosystem includes multi-asset shielded transfers through a versatile multi-asset shielded pool derived from the Sapling circuit. Offering comprehensive support for the Inter-Blockchain Communication (IBC) protocol and integrated Ethereum bridge, Namada introduces a robust proof-of-stake system with cubic slashing, stake-weighted governance signaling, and a novel dual public goods funding approach. Users are incentivized through native protocol tokens (NAM) for their contributions to the shielded set, reinforcing a commitment to data protection.

The report is focused on three parts of this audit: ABCI, Replay protection and fee and gas metering.

ABCI

Namada has developed a custom ABCI (Application Blockchain Interface) implementation built upon CometBFT v0.37. They have implemented a wrapper around CometBFT to incorporate ABCI++ features. The system comprises a shell that implements ABCI++ and a shim that wraps it. The shim is utilized to translate between the ABCI interface of the current CometBFT version and the shell's interface. For instance, the shim implements BeginBlock, DeliverTx, and EndBlock functions, which gather data and, ultimately, invoke the shell's FinalizeBlock function at the EndBlock stage.

In this part of the code, the prepare proposal is also implemented, where the selected proposer decides on the transactions to be included in the block, and the process proposal through which other validators verify that proposal. Since there are Protocol and Wrapper transactions (which include inner transactions), their distribution in the block is managed through a block allocator.

Additionally, a mechanism for vote extensions has been implemented, although it was not within the scope of the audit.

Replay Protection

Replay protection is safeguarding against malicious actors attempting replay attacks, where previously executed transactions are resubmitted to the ledger. Such attacks disrupt the intended state of the system, inflicting economic harm on the fee payer by causing them to incur additional costs. If the original transaction involved value transfer, the sender may suffer further financial loss. Given that attackers can exploit well-formatted transactions without modification, replay attacks pose a significant threat and must be effectively countered.

Namada's implementation of replay protection mitigates the risk posed by replay attacks by preventing the execution of duplicate transactions. Leveraging the encrypted and authenticated communication channels provided by CometBFT, Namada ensures secure transmission of transactions between nodes. Transactions consist of two components: a WrapperTx and an inner Tx, facilitating the exchange of data defining ledger state transitions within the Namada protocol.

The implementation of replay protection in Namada comprises several key components aimed at replay attacks and ensuring the integrity of transactions. These include a robust hash-based solution for both WrapperTx and EncryptedTx, mitigating the risk of attackers extracting and replaying inner transactions. Additionally, Namada implements checks within `mempool_validate` and `process_proposal` to verify the uniqueness of transactions, rejecting any duplicates. The inclusion of a `ChainId` identifier and transaction expiration further enhances security by binding transactions to specific forks and imposing a time limit for execution, respectively.

Fee and gas metering

Namada's fee system plays a crucial role in maintaining the integrity and efficiency of its ledger. Transaction fees serve a dual purpose: to allocate block space and gas efficiently and to incentivize block producers to prioritize transactions. These fees can be paid in any fungible token listed on Namada's whitelist, with minimum fee rates set by governance. Additionally, when utilizing the shielded pool, transactions can unshield tokens to cover the required fees.

The token whitelist, comprising token identifiers and minimum gas prices, is periodically updated through governance proposals. All fees collected are directly paid to the block proposer, ensuring an incentive-compatible system. Namada's upfront fee payment mechanism prevents denial-of-service attacks by ensuring transactions pay for resources upfront, enhancing block inclusion efficiency and discouraging spam.

The Fee field within the `WrapperTx` struct encapsulates the payment data, including the fee amount and token address. The fee payer, typically the signer of the wrapper transaction, specifies the token and amount, meeting the minimum gas price set by the whitelist. These parameters incentivize block proposers to prioritize transactions. Validators validate these parameters during block construction, ensuring the integrity of the process.

Gas accounting in Namada is a process designed to manage the utilization of two critical resources within a block: gas and space. Each transaction incurs a fixed amount of gas per byte to address the space limit, ensuring efficient allocation. Gas limits are calculated for transactions and validity predicates, considering factors such as the runtime cost of wasm code and validity predicate requirements. To prevent transactions from exceeding their gas limits, a gas counter is embedded within each transaction and VP, allowing real-time monitoring of gas consumption. This mechanism enables immediate validation against the declared GasLimit set in the corresponding wrapper transaction.

In this audit, the fee payment system and gas metering were followed through `prepare_proposal`, `process_proposal` and `finalize_block`.

Threat Analysis

ABCI

1. The set of transactions in Namada’s `prepareProposal` response exceeds the maximum amount of bytes

Consequences:

- It violates Requirement 2 [`PrepareProposal` , tx-size] of the ABCI++ 0.37 specification: when p's Application calls `ResponsePrepareProposal` , the total size in bytes of the transactions returned does not exceed `RequestPrepareProposal.max_tx_bytes` .
- It compromises liveness as the consensus engine will reject proposals from honest validators that exceed the maximum allowed.

Conclusion: The implementation suffers from this threat, see Inconsistent Block Size Limit Handling in Proposal Preparation

2. Honest validators reject proposals from honest proposers

Consequences:

- It violates Requirement 3 [`PrepareProposal` , `ProcessProposal` , coherence] of the ABCI++ 0.37 specification: for any two correct processes p and q, if q's CometBFT calls `RequestProcessProposal` on up, q's Application returns `Accept` in `ResponseProcessProposal` .
- It compromises liveness as validators will reject proposals coming from honest proposers.

Conclusion: The implementation does not suffer from this threat. It has been verified that every proposal generated from `prepareProposal` will be guaranteed to be accepted by `processProposal` . Some checks are performed by the proposer in the prepare proposal phase, while others are done when adding to the mempool. Certain validations related to time (expiration, epoch) are correctly verified in the mempool through `Recheck` .

3. Namada’s `processProposal` is non-deterministic, i.e., its result does not depend exclusively on the proposal and the committed blockchain state

Consequences:

- It violates Requirement 4 [`ProcessProposal` , determinism-1]: of the ABCI++ 0.37 specification: `ProcessProposal` is a (deterministic) function of the current state and the block that is about to be applied. In other words, for any correct process p, and any arbitrary block u, if p's CometBFT calls `RequestProcessProposal` on u at height h, then p's Application's acceptance or rejection exclusively depends on u and sp,h-1.
- It compromises liveness as validators will reject proposals coming from honest proposers.

Conclusion: Upon analysis of `ProcessProposal` , it has been determined that it executes deterministically, meaning that the outcome depends exclusively on the proposal and the committed blockchain state.

Additionally, it is important to note that functions `validate_eth_events_vext` , `validate_bp_roots_vext` , and `validate_valset_upd_vext` , located on the Ethereum bridge, were not extensively analyzed within this determinism assessment. However, it was assumed that these functions are deterministic.

4. Namada’s `prepareProposal` and/or `processProposal` modify the blockchain state

Consequences:

- It violates Requirement 9 [all, no-side-effects]: of the ABCI++ 0.37 specification: p’s calls to `RequestPrepareProposal` , `RequestProcessProposal` at height h do not modify sp,h-1.
- It compromises safety, as at `prepareProposal` and `processProposal` , the proposal has not yet been decided, so it may need to be rolled back.

Conclusion: Through analysis, it has been determined that the requirement is satisfied. The original states are only used for retrieving information, while all modifications are performed on a temporary object (`temp_state = self.state.with_temp_write_log();`) which is re-instantiated in each `prepareProposal` and `processProposal` executions.

5. Namada’s `beginBlock` , `deliverTx` and `endBlock` are non-deterministic, i.e., its result does not depend exclusively on the proposal and the committed state

Consequences:

- It violates Requirement 11 [`BeginBlock` - `DeliverTx` - `EndBlock` , determinism-1] of the ABCI++ 0.37 specification: for any correct process p, sp,h exclusively depends on sp,h-1 and vp,h.
- It violates Requirement 12 [`BeginBlock` - `DeliverTx` - `EndBlock` , determinism-2] of the ABCI++ 0.37 specification: for any correct process p, the contents of Tp,h exclusively depend on sp,h-1 and vp,h.
- It may violate safety as validators may end up with different blockchain states after processing the same set of blocks in the same order.

Conclusion: This threat has been analyzed specifically for the requests `BeginBlock` , `DeliverTx` , and `EndBlock` within the `run` function in `abcipp_shim.rs` (/shims/abcipp_shim.rs#L105-L174).

- `BeginBlock` **and** `DeliverTx` : These requests simply store data received from CometBFT, which is part of the proposal. Adding events to the response in `DeliverTx` does not affect determinism.
- `EndBlock` : In this phase, all data is obtained from the proposal and the current state (`block_time` , `block_proposer` , etc.). All subsequent functions called are deterministic (`process_tx` , `swap` , `zip` , `push` , etc.) and are invoked exclusively with data from the proposal or the state.

Therefore, based on the analysis, it can be concluded that the operations performed within `BeginBlock` , `DeliverTx` , and `EndBlock` requests do not compromise determinism as they strictly adhere to using proposal data or current state information.

6. Namada persists blockchain state before commit is called

Consequences: It may violate safety as validators may end up with different blockchain states after processing the same set of blocks in the same order. This is because each validator may execute a different set of ABCI calls. The consensus engine only guarantees that all validators eventually commit the same set of blocks in the same order, but each validator may execute different sets of `processProposal` calls.

Conclusion: The implementation does not suffer from this threat. Any changes made to transaction-related states will be rolled back in case of an error during transaction execution. On the other hand, information related to the block (block height, epoch, block hash, etc.) must be stored regardless of the success of transaction execution, which is correctly implemented.

7. The shim implementation of `finalizeBlock` from the CometBFT 0.37 ABCI API is incorrect

Consequences: Unknown/unpredictable behavior.

Conclusion: The implementation does not suffer from this threat, but during the analysis, we caught this related informational finding:Error Propagation Issue in EndBlock Handling

8. Malicious proposers can censor non-expired transactions by biasing the block time

Consequences: Malicious validators can censor valid transactions at will.

Conclusion: Through the analysis of this threat, it has been determined that a malicious proposer does not have the ability to censor non-expired transactions.

On CometBFT, the block time is generated and distributed to validators through requests, both in the prepare proposal phase and the process proposal phase. Furthermore, any attempts by a proposer to manipulate the time would be ineffective, as validators verify the expiration of transactions during the process proposal phase using the time provided by CometBFT through requests.

The expiration time of a transaction is located in the transaction header and is signed by the transaction creator. Since the transaction signature is verified during the process proposal phase, it is not feasible for a malicious proposer to alter the expiration time of a transaction.

This analysis provides assurance that the integrity of transactions remains protected from malicious attempts at censorship by proposers.

9. Proposals generated by honest proposers include invalid wrapper transactions

Consequences: It compromises liveness, as honest proposers may get their proposals rejected because of this.

Conclusion: The implementation does not suffer from this threat. The proposer in the prepare proposal phase checks the following aspects related to wrapper transactions: signature validity, ensuring that the gas limit is below the block gas limit, correctness of the `ChainId`, ensuring that the transaction has not expired, acceptance of tokens for fee payment and possession of the minimum required amount. If the transaction includes an unshielding transaction, it is considered valid.

10. Validators accept proposals that include invalid wrapper transactions

Consequences: Malicious proposers can waste block space by including invalid wrapper transactions.

Conclusion: The implementation does not suffer from this threat. Validators in the process proposal phase verify the following aspects related to wrapper transactions, and in case of invalidity of any wrapper transaction, they will reject the proposal: signature validity, ensuring that the gas limit is below the block gas limit, correctness of the `ChainId`, ensuring that the transaction has not expired, acceptance of tokens for fee payment and possession of the minimum required amount. If the transaction includes an unshielding transaction, it is considered valid.

11. Proposals generated by honest proposers include invalid inner transactions

Consequences: It compromises liveness, as honest proposers may get their proposals rejected because of this.

Conclusion: The implementation does not suffer from this threat. In the prepare proposal phase, the proposer verifies the following aspects related to inner transactions: it passes the replay protection check, and it is one of the predefined transactions. If an error occurs, transaction will not be included in the proposal.

12. Validators accept proposals that include invalid inner transactions

Consequences: Malicious proposers can waste block space by including invalid inner transactions.

Conclusion: The implementation does not suffer from this threat. Validators in the process proposal phase verify the following aspects related to inner transactions, and in case of invalidity of any inner transaction, they will reject the proposal: it passes the replay protection check, and it is one of the predefined transactions.

Replay protection

1. A committed inner transaction can be re-wrapped and replayed

Consequences: A user's transaction can be replayed an unbounded number of times. If for instance, the inner transaction is a transfer, by replaying it, the malicious user could take all the funds from the source account.

Conclusion: The implementation does not suffer from this threat. If an inner transaction is committed, i.e., included in a decided block, its hash is registered unless the execution of the associated wrapper fails or its execution fails due to gas consumption or because it is missing code section. This means that if the transaction is successfully executed, its hash is registered. If a successfully executed inner transaction is resubmitted, honest validators will reject the proposal in which it is included due to the replay protection checks in `processProposal`.

2. A committed wrapped transaction can be replayed

Consequences: A wrapped transaction can be replayed an unbounded number of times. If for instance, the inner transaction is a transfer, by replaying it, the malicious user could take all the funds from the source account. The consequences are limited to the time when the transaction expires, which cannot be manipulated as in Threat #1.

Conclusion: The implementation does not suffer from this threat. If wrapper transaction is committed, i.e., included in a decided block, either its hash or the hash of the associated inner transaction is registered unless the wrapper execution fails. Thus, if a successfully executed wrapper transaction is resubmitted, honest validators will reject the proposal in which it is included due to the replay protection checks in `processProposal`.

3. If a block includes duplicate inner transactions, validators execute them more than once

Consequences: A user transaction is executed multiple times. For instance, if the inner transaction is a transfer, the source account may transfer more tokens than the transaction's user intended.

Conclusion: The implementation does not suffer from this threat. A proposal may contain an inner transaction multiple times. Nevertheless, the execution of a proposal in `finalizeBlock` will prevent the execution of duplicates. Intuitively, this is prevented because the first time an inner transaction is executed, the inner transaction hash would be registered in the replay protection write log. Thus, when a validator attempts to execute the same inner transaction, the replay protection checks in `apply_wasm_tx` will detect the replay attack and abort the execution of the transaction.

4. Proposed blocks can contain duplicate wrapped transaction

Consequences: A user transaction is executed multiple times. For instance, if the inner transaction is a transfer, the source account may transfer more tokens than the transaction's user intended.

Conclusion: The implementation does not suffer from this threat. Proposal with duplicate wrapper transactions are rejected by `processProposal` .

5. Hash register grows unboundedly

Consequences: Eventually, validators run out of storage.

Conclusion: The implementation suffers from this threat, see The replay protection hash register is never garbage collected.

6. The replay protection mechanism prevents users from submitting identical transactions

Consequences: The replay protection system produces false positives. As a consequence, a valid user transaction may be identified as a duplicate by the replay protection system despite not having been executed before.

Conclusion: The implementation does not suffer from this threat. This is prevented because the `timestamp` field in transactions makes transactions from the same user unique with a very high probability. To break this, a user would have to generate twice the same transaction in the same millisecond.

7. The hash register includes hashes of transactions that have not been committed

Consequences: a valid user transaction may be identified as a duplicate by the replay protection system despite not having been executed before.

Conclusion: The implementation does not suffer from this threat. The hash register may include transactions that have not been successfully executed in the following cases:

- The execution of the wrapper failed, in which case a wrapper's hash is registered.
- The inner transaction has not been accepted and it did not fail due to an invalid signature.
- The execution of the inner transaction failed due to anything but gas consumption or section commitment. For gas consumption, the hash is not registered because it is fixable by rewrapping the transaction and increasing the gas limit. Similarly for section commitment, it is fixable by resubmitting the transactions with the same header but including the code section that was likely missing before.

In all cases, it is reasonable to store the hashes to prevent adding such transactions in future blocks to optimize block space given that those transactions are likely to fail execution as they are.

8. Fork replay attacks are possible

Consequences: Transactions from another branch (fork) can be executed, i.e., replayed, which is unintended by the original user.

Conclusion: The implementation does not suffer from this threat. Transactions include a `chain_id` field that uniquely identifies to which chain (or fork) the transaction is intended to be submitted. Validators check in `processProposal` that the transaction's `chain_id` matches the ledger's `chain_id` . This prevents fork replay attacks.

Fees and gas metering

1. Fees are paid in a token that is not a member of a whitelist controlled by Namada governance

Consequences: Users can execute transactions paying in a token without any value.

Conclusion: The implementation does not suffer from this threat. To control how fees are paid in terms of allowed tokens, the provided token is checked in `mempool_fee_check.` The code is checking if there is associated gas cost for the provided token in the storage. If not, the system returns an error. This kind of verification is performed also in `prepare_proposal` and `process_proposal`.

2. A transaction can pay less than the minimum fee rate for the given allowed token

Consequences: Users can execute transactions at a discount price.

Conclusion: The implementation does not suffer from this threat. Similarly to previous threat, a process of retrieving allowed token includes the checks for minimum gas price being paid. Being present in both `process_proposal` and `prepare_proposal` it is blocking any transactions without paying minimum required price.

3. The gas metering does not add up all the gas used

Consequences: Users can execute transactions at a discount price by picking a lower gas limit.

Conclusion: The implementation does not suffer from this threat. Gas meters were updated correctly without unnecessary restarts. Gas consuming for storage reads, compiling, ibc actions etc. are measured according to Namada's custom defined prices and added to final count.

4. Early verification (before they pay the fees) is too expensive

Consequences: Malicious users can easily spam validators by forcing them to do an expensive verification without paying any fee.

Conclusion: The code does not suffer from this threat. However, during he analysis we found that some of the validations could be considered redundant. See informational findingStreamlining Validation Logic Across Proposal Lifecycle .

5. Early verification is unable to detect invalid transactions

Consequences: Due to inability to early detect transactions that should not end up in the block, block space is wasted.

Conclusion: The implementation suffers from this threat. It is possible to prevent scenario that is leading to liveness issue by adding an early verification, see Block could be rejected due to proposer and validators using different minimum gas price .

6. The gas metering system makes transaction execution too expensive

Consequences: Executing a transaction is too expensive, which may discourage user from using Namada.

Conclusion: The implementation does not suffer from this threat. Namada relies on their benchmarking in several places when it comes to charging gas, and also does some scaling to keep the gas in expected range and be able to charge reasonable fees. Overpriced execution was not detected.

7. The gas metering system makes transaction execution too cheap

Consequences: Malicious users can easily spam and DoS Namada if the execution is too cheap.

Conclusion: The implementation does not suffer from this threat. No underpriced execution was detected.

8. A malicious user can overflow the gas counter by submitting a high gas-consuming transaction

Consequences: Validators may crash, or the user is able to execute a high gas-consuming transaction for a discount price.

Conclusion: The implementation does not suffer from this threat. All the operations of increasing consumed gas are executed using checked arithmetic. This is preventing any overflows in gas calculations.

9. A transaction includes a valid unshielding transaction that does not unshield enough funds to pay the fee

Consequences: Users can execute transactions at a discount price by unshielding less funds than required by the fee.

Conclusion: The implementation does not suffer from this threat. The wrapper transaction is carrying optional special data about unshielding. This data gets checked against descriptions limit to prevent DoS attacks. An unshielding transaction is then created and applied using `apply_wasm_tx` on a temporary state where it should transfer funds for fee payment to the fee payer. On successful execution of this, another safety check is the `check_fees` where the balance of the fee payer is checked once more to determine there is enough funds for fee payment.

10. Fees for multiple transactions inside a single block could not be paid by the same fee payer

Consequences: One or more of the transactions drain the funds of the fee payer making other transactions unable to pay for the fee.

Conclusion: The implementation does not suffer from this threat. The process of determining fee payer's balance is done in `prepare_proposal` , `process_proposal` and `finalize_block` in a function `transfer_fee` . It includes changing one state (temporary in case of prepare and process proposal) while iterating over transactions, which means it is accounting for the scenario that multiple transaction fees are paid from single fee payer.

Findings

Title	Type	Severity	Impact	Exploitability
Block could be rejected due to proposer and validators using different minimum gas price	IMPLEMENTATIO N	4 CRITICAL	3 HIGH	3 HIGH
Inconsistent Block Size Limit Handling in Proposal Preparation	IMPLEMENTATIO N	4 CRITICAL	3 HIGH	3 HIGH
The replay protection hash register is never garbage collected	IMPLEMENTATIO N	2 MEDIUM	3 HIGH	1 LOW
Replay protection can be significantly simplified	PROTOCOL	0 INFORMATIONAL	0 NONE	0 NONE
https://informalsystems.atlassian.net/wiki/spaces/A2Q1/pages/369950801	IMPLEMENTATIO N	0 INFORMATIONAL	0 NONE	0 NONE
Error Propagation Issue in EndBlock Handling	IMPLEMENTATIO N	0 INFORMATIONAL	0 NONE	0 NONE
Code Optimization Recommendations	IMPLEMENTATIO N	0 INFORMATIONAL	0 NONE	0 NONE
Streamlining Validation Logic Across Proposal Lifecycle	IMPLEMENTATIO N	0 INFORMATIONAL	0 NONE	0 NONE

Block could be rejected due to proposer and validators using different minimum gas price

Project	**Namada 2024 Q2**
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- crates/apps/src/lib/node/ledger

Description

In the `prepare_proposal` and `process_proposal` functions, there is a significant difference in how the minimum gas price (`minimum_gas_price`) is determined. While both functions share a similar procedure for setting this value, `prepare_proposal` accounts for possible local validator configuration, whereas `process_proposal` relies only on consensus parameters. This poses a risk of liveness issues within the system.

Problem Scenarios

Consider a scenario where a proposer reads the minimum gas price from its local configuration, while followers base their votes on the minimum gas price specified in the consensus parameters. If the minimum gas price in the proposer's configuration is lower than the one specified in the consensus parameters, the votes from other validators would likely result in rejection.

Recommendation

Ensure consistency in how the minimum gas price is determined across all functions involved in proposal processing. This includes both `prepare_proposal` and `process_proposal` functions.

Consider implementing constraints to enforce that the minimum gas price specified in the local configuration is always higher than or equal to the minimum gas price specified in the consensus parameters.

Inconsistent Block Size Limit Handling in Proposal Preparation

Project	**Namada 2024 Q2**
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- crates/apps/src/lib/node/ledger

Description

The process of preparing proposals involves the BlockAllocator, which sets the maximum size of all transactions. This maximum size, determined by the parameter `max_proposal_bytes` in the BlockAllocator configuration (/block_alloc.rs#L141), should align with the `max_tx_bytes` parameter in the `RequestPrepareProposal`. However, there lacks a validation step to ensure this alignment, potentially leading to discrepancies between these parameters over time.

Problem Scenarios

The absence of consistency checking between `max_proposal_bytes` and `max_tx_bytes` poses several risks. Firstly, it violates Requirement 2 [PrepareProposal, tx-size] of the ABCI++ 0.37 specification, which mandates that the total size of transactions in a proposal does not exceed `max_tx_bytes` when calling `ResponsePrepareProposal`. Secondly, such discrepancies compromise liveness, as proposals from honest validators exceeding the maximum allowed would be rejected by the consensus engine.

Recommendation

To address these issues, it would be desirable to implement a validation step to ensure that `max_proposal_bytes` does not exceed `max_tx_bytes`, or that the total size of transaction bytes in the proposal does not exceed the `max_tx_bytes` specified in the `RequestPrepareProposal`.

The replay protection hash register is never garbage collected

Project	**Namada 2024 Q2**
Type	**IMPLEMENTATION**
Severity	**2 MEDIUM**
Impact	**3 HIGH**
Exploitability	**1 LOW**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- crates/apps/src/lib/node/ledger

Description

The hashes of executed transactions are stored in a storage hash register. This allows Namada to prevent replay attacks: if an already executed transaction is resubmitted, Namada guarantees that its hash is in the hash register and is thus able to reject block proposals that include duplicate transactions. Nevertheless, the hash register is never garbage collected.

Problem Scenarios

The hash register grows unboundedly as Namada produces blocks. This is a problem as it can exhaust nodes' storage and possibly make operations on the hash register more expensive over time.

Recommendation

We recommend removing expired transactions from the hash register when a block is committed. This would require storing the hashes in a secondary data structure indexed by expiration time (or a similar approach) to avoid iterations over the whole hash register to delete expired transactions.

Replay protection can be significantly simplified

Project	**Namada 2024 Q2**
Type	PROTOCOL
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- crates/apps/src/lib/node/ledger

Description

When a block `b` is committed, the hashes of the transactions included in a block are stored for replay protection. The hashes are stored under the `last` prefix in the store. When block `b+1` is committed, all the hashes stored under the `last` prefix are finalized. To finalize a hash requires three storage operations:

- Delete the hash under the `last` prefix
- Store the hash under the `all` prefix
- Store the hash under the `buffer` prefix

Such an intricate mechanism is in place to support a one-block roll-back. We argue that the implementation could be simplified.

Recommendation

We recommend simplifying the implementation as follows:

- Remove replay protection write log states: if a hash is in the write log replay protection map, it means that has to be written in storage.
- Instead of writing wrapper hashes in `apply_wrapper_tx`, do it selectively in `finalize_block` directly depending on the result of `dispatch_tx`. For instance, write it if `dispatch_tx` returns a `WrapperRunnerError` error. This way, one can get rid of the `DELETE` write log state: no need to remove wrapper hashes.
- The `commit_write_log_block` function would look as follows:
    - First, remove all hashes under the `buffer` prefix.

    - Store all hashes in the write log under both `all` and `buffer` prefixes.
- The replay protection checks in prepare and process proposal only need to check for hashes under the `all` prefix.
- On rollback, iterate over the hashes under the `buffer` prefix and remove them from the `all` prefix.

Incorrect Error Code Mapping in check_proposal_tx Description

Project	**Namada 2024 Q2**
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /crates/apps/src/lib/node/ledger
- /crates/tx/src/data

Description

The `check_proposal_tx` function contains error codes that are not correctly mapped to their corresponding IDs in the description comment. This discrepancy can lead to confusion and potential misinterpretation of error conditions. For example, the error code `WasmRuntimeError` is listed as 3 in the function's description comment, but it is defined as 1 in the `ResultCode` enum, not 3 as indicated in the comment. Additionally, some error codes are not listed at all in the function's description comment, leading to incomplete documentation.

Problem Scenarios

Developers relying on the error code documentation within the description comment of the `check_proposal_tx` function may encounter confusion considering discrepancies between the documented error codes and their actual definitions.

Recommendation

To ensure consistency and accuracy, it is recommended to align the error code documentation in the description comment of the `check_proposal_tx` function (/process_proposal.rs#L174-L184) with the definitions in the `ResultCode` enum (/mod.rs#L55-L86).

Error Propagation Issue in EndBlock Handling

Project	**Namada 2024 Q2**
Type	**IMPLEMENTATION**
Severity	**0 INFORMATIONAL**
Impact	**0 NONE**
Exploitability	**0 NONE**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- /crates/apps/src/lib/node/ledger

Description

`BeginBlock`, `DeliverTx`, and `EndBlock` functions collect data and execute everything together in `EndBlock`, simulating the behavior of `FinalizeBlock`. According to the CometBFT v0.37 documentation, `EndBlock` on CometBFT does not expect any errors. However, in the code, errors are packed in the `EndBlock` response.

Problem Scenarios

In the `finalize_block` function, errors are propagated in several places within the `finalize_block` function and its sub-systems (/finalize_block.rs#L105-L119, /finalize_block.rs#L123). Errors in any of these places can be propagated to `EndBlock`, contrary to the expected behavior described in the documentation.

Recommendation

To ensure consistency with the expected behavior described in the documentation review the error-handling mechanism in the `finalize_block` function and its sub-systems. Errors should not be propagated to `EndBlock` to avoid potential crashes or disruptions in the system's operation. However, it's essential to consider the severity of the error; in some cases, an error and subsequent crash may be justified, especially when the system encounters a critical issue that cannot be safely handled.

Code Optimization Recommendations

Project	**Namada 2024 Q2**
Type	**IMPLEMENTATION**
Severity	**0 INFORMATIONAL**
Impact	**0 NONE**
Exploitability	**0 NONE**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- crates/apps/src/lib/node/ledger

Description

Upon reviewing the codebase, several optimizations were identified that could enhance the efficiency and readability of the code.

Problem Scenarios

1. The `validate_tx` function is redundantly called twice in the `check_proposal_tx` function, during process proposal, leading to unnecessary overhead.
   1st: /process_proposal.rs#L249-L256
   2nd: /process_proposal.rs#L265-L270
2. The previously defined `inner_tx_hash` should be used instead of `&tx.raw_header_hash()`. mempool_validation()
3. Move parameter obtaining from storage outside loop iterations to minimize unnecessary storage calls. In this way, the number of calls to storage would be reduced. /governance.rs#L79
4. Since the `validate_wrapper_bytes` function is specific to Wrapper transactions, the transaction type check should be performed at the beginning of the function. If the transaction is not a WrapperTx, an error should be immediately returned. /prepare_proposal.rs#L286

Recommendation

To improve performance, it is recommended to implement the suggestions outlined in the Problem Scenarios section.

On Streamlining Validation Logic Across Proposal Lifecycle

Project	**Namada 2024 Q2**
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /crates/apps/src/lib/node/ledger
- /crates/namada/src/vm/wasm

Description

Any stateless validation done when adding a transaction to the mempool does not need to be done again in prepare proposal. This is because a proposal from an honest proposer only includes transactions from its mempool. It is then guaranteed that any transaction in such a proposal has passed all those checks.

The same reasoning could be applied to process proposal and finalize block: under the assumption that `2f+1` of the voting power is controlled by honest validators, CometBFT guarantees that any block in a finalize block call has been validated by at least one honest validator via process proposal. Nevertheless, if a larger set of validators is compromised, this may not be true anymore. In such a case, one may want to replicate any validity checks done in process proposal in finalize block to guarantee the validity of histories under all circumstances.

We have identified one redundant validity check in prepare proposal and one in finalize block.

Problem Scenarios

- Redundant signature check in mempool validation (code) and prepare proposal (code),
- Redundant signature check in process proposal (code) and finalize block (code).

Recommendation

We recommend removing the duplicated signature check from prepare proposal. Regarding the checks in finalize block, we recommend proceeding differently depending on whether the team wants to guarantee the validity of histories under all circumstances:

- If required, we recommend making sure that all validity checks in process proposal are replicated in finalize block.
- If not required, we recommend removing the duplicated checks from finalize block.

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.

Impact Score	**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)
🔵 **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:

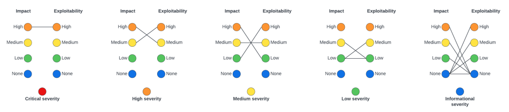
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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)

Exploitability Score	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.



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.

Severity Score	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

Severity Score	Examples
🟢 **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