



Security Audit Report

Namada Governance & PGF

Authors: Ivan Gavran, Ivan Golubovic, Manuel Bravo, Tatjana Kirda

Last revised 11 December, 2024

Table of Contents

Audit Overview	1
The Project	1
Audit Timeline	1
Conclusion	1
Audit Dashboard	2
Target Summary	2
Engagement Summary	2
Severity Summary	2
System Overview	3
Governance	3
Public Goods Funding (PGF)	4
Threat Analysis	5
Safety Properties	5
Liveness Properties	15
Findings	18
Malicious user could prevent honest users from installing valid governance proposals by not increasing the governance counter	20
A governance proposal may prevent creation of future proposals	22
Adding an incomplete proposal may cause panic in finalize_block	26
Proposal type key modification	29
Proposal funds key modification	31
Some checks for governance-vetted updates are justified and beneficial	33
Voting starting epoch starts only strictly after the proposal epoch	34
There's no need to deal with multiple governance proposals being installed in parallel	35
A wrong check for max_proposal_period	36
Max Proposal Latency Parameter Not Included in the is_parameter_key function	37
Some VP-functions are rendered trivial by their reachability conditions	39
Miscellaneous Code Findings	41
Disclaimer	43
Appendix: Vulnerability Classification	44

Impact Score	44
Exploitability Score	44
Severity Score	45

Audit Overview

The Project

In November and December 2024, the Anoma Foundation engaged [Informal Systems](#) to work on a partnership and conduct a security audit of the following items in Namada's software:

1. Governance crate including PGF (Public Goods Funding): [namada/crates/governance](#)
2. Governance and PGF transactions:
 - a. [namada/wasm/tx_update_steward_commission](#)
 - b. [namada/wasm/tx_resign_steward](#)
 - c. [namada/wasm/tx_init_proposal](#)
3. Related entrypoints in [namada/crates/apps_lib/src/cli/client.rs](#) as well as `finalize_block` handler in [namada/crates/node/src/shell/finalize_block.rs](#).

The audit was performed from November 18th, 2024 to December 9th, 2024 by the following personnel:

- Ivan Gavran
- Ivan Golubovic
- Manuel Bravo
- Tatjana Kirda

Relevant Code Commits

The audited code was from:

- commit hash 207fe50a8fab63e0c84a0e6b4a4ef31de7f4ce98 (tag `v0.45.1`).

Audit Timeline

- 19.11.2024. Kick-off meeting and code walkthrough by the Namada team.
- 25.11.2024. First weekly sync meeting. The auditing team has shared issues that surfaced.
- 02.12.2024. Second weekly sync meeting. The auditing team has shared issues that surfaced.
- 09.12.2024. Closure meeting. The draft report was shared, and all findings surfaced were shared.

Conclusion

After conducting a thorough review of the project, we found it to be carefully designed and generally well-implemented.

The current Namada's deployment only allows a set of whitelisted transactions to be executed. Our analysis concludes that there are no major issues in the governance and PGF modules under such conditions. We have also considered the case when Namada executes arbitrary transactions, as this is the roadmap. Our analysis concludes that the code is not ready for such a challenging deployment: we found some problems that if left unattended, would violate both safety and liveness properties (more on them in the Findings section).

Audit Dashboard

Target Summary

- **Type:** Protocol and Implementation
- **Platform:** Rust
- **Artifacts** audited over the fixed `207fe50` commit hash.
 - `namada/crates/governance`
 - `namada/wasm/tx_update_steward_commission`
 - `namada/wasm/tx_resign_steward`
 - `namada/wasm/tx_init_proposal`
 - Related entrypoints in `namada/crates/apps_lib/src/cli/client.rs` as well as `finalize_block` handler in `namada/crates/node/src/shell/finalize_block.rs`.

Engagement Summary

- **Dates:** 18.11.2024. - 09.12.2024.
- **Method:** Manual code review, protocol analysis

Severity Summary

Finding Severity	#
Critical	5*
High	0
Medium	1
Low	1
Informational	5
Total	12

[*] - only for arbitrary transactions

System Overview

Governance

The Namada governance mechanism serves at least two purposes: to allow users and operators to upgrade the protocol dynamically and to enable social coordination.

Any user account can submit a governance proposal on-chain and optionally some code for the protocol to execute if the proposal is accepted. A proposal without code is of type `Default` and a proposal with code is of type `DefaultWithWasm`. Validators and delegators with active bonds can vote on the proposals with voting power proportional to their bonded token amount.

The execution of a governance proposal is valid if:

- The proposal id is unique: there is no previously installed proposal with the same proposal id
- It should write into storage all the required fields encoding the proposal's id into the fields' keys. The required fields are: `content`, `author`, `committing_epoch`, `voting_start_epoch`, `voting_end_epoch`, and `activation_epoch`.
- If the proposal includes a non-empty `data` field, then its content depends on the proposal type:
- `content` must satisfy the following:
 - `size(content) <= max_proposal_content_size`
- `author` must satisfy the following:
 - It is an existing Namada user
 - Its identity can be verified
- `voting_start_epoch` must satisfy the following:
 - `voting_start_epoch >= current_epoch`
 - `voting_end_epoch - voting_start_epoch >= min_proposal_voting_period`
 - `activation_epoch - voting_start_epoch <= max_proposal_period`
 - `voting_start_epoch - current_epoch <= max_proposal_latency`
- `voting_end_epoch` must satisfy the following:
 - `voting_end_epoch >= current_epoch`
 - `voting_end_epoch - voting_start_epoch >= min_proposal_voting_period`
 - `activation_epoch - voting_end_epoch >= min_proposal_grace_epochs`
- `activation_epoch` must satisfy the following:
 - `activation_epoch >= current_epoch`
 - `activation_epoch - voting_start_epoch <= max_proposal_period`
 - `activation_epoch - voting_end_epoch >= min_proposal_grace_epochs`
- The user that submits the proposal transfers at least `min_proposal_fund` to the governance internal address.
- Only a proposal of type `DefaultWithWasm` can write the `PROPOSAL_CODE` key.
- The length of the WASM code written to the proposal code key by a `DefaultWithWasm` proposal must not exceed the maximum allowed size defined by `max_proposal_length`

The execution of a vote transaction is valid if:

- It votes for an existing proposal
- The voter is either an existing validator or delegator with active bonds at the current epoch.

- The vote is being executed within the voting period
- If the voter is a validator, the vote is being executed within the 2/3 of the voting period

Public Goods Funding (PGF)

Namada funds public goods by regularly minting NAM tokens into an on-chain PGF account and allocating certain amounts for certain entities. The inflation rate is a constant parameter `pgf_inflation_rate` that is mutable by governance.

PGF builds on top of governance as follows. The entities that receive such funding are approved either via governance (via a `PGFPayment` proposal) or by a multi-signature account known as a "public goods steward".

Stewards must also be elected with a governance proposal (via a `PGFSteward` proposal) and are meant to be trusted authorities for responsibly identifying good candidates to receive public goods funding. As such, a steward may submit a funding proposal that does not need to receive any positive votes to be passed. There is also a mechanism in place to veto stewards.

In exchange for their work, stewards receive rewards at the end of each epoch. Since in most cases it is assumed that a steward is a multi-signature account, Namada allows stewards to distribute their rewards among multiple accounts.





Specific validity conditions for PGF governance proposals:

- `PGFSteward` proposals must also ensure:
 - Add at most one steward
 - Include no more than `MAX_PGF_ACTIONS` actions
 - If a steward is being added, only he can be the author
 - Uniqueness of addresses in actions
- `PGFPayment` proposals must also ensure:
 - Uniqueness of target for each action type: add/remove continuous, add retro
 - No continuous target is removed and added within the same transaction

Finally, stewards can resign or change their reward distribution at will by submitting the appropriate transactions. A reward distribution is valid if it does not exceed the limit of addresses (`REWARD_DISTRIBUTION_LIMIT`), each distribution is between 0 and 1, and the total sum is less or equal to 1.

Threat Analysis

In our threat analysis, we start by defining a set of properties required for the correctness of the governance and PGF modules in Namada. We separate them into safety and liveness properties. For each property, we define one or more threats. We then analyze them individually to see if they could be violated, resulting in the findings presented in the Findings section. Note that each threat is marked with one of the following:


-  Indicates that the threat does not materialize, and no vulnerabilities have been identified in the current implementation, or
-  Indicates that the threat is valid, and vulnerabilities have been identified, or
-  /  Indicates that vulnerabilities exist, but they can only be exploited under the assumption arbitrary transactions are used. Scenarios restricted to whitelisted transactions do not result in exploitable conditions.

Let us first start with a set of definitions:

- A proposal can be any of the following states:
 - Installed: a transaction with the proposal is included in a decided block and accepted by the relevant validity predicates, i.e., the proposal is persisted in Namada's state.
 - Accepted: it has enough valid votes by the end of the voting period.
 - Executed: Namada has executed it during finalize block.
- The definitions of validity, e.g., proposal and vote validity, are in [System Overview](#).

Safety Properties

1. A governance proposal is accepted only if it has enough valid votes


Threat 1.1: A malicious user sends messages in the name of other users and Namada does not authenticate votes properly. 

Conclusion: The threat does not hold. The `is_valid_vote_key` function ensures that voters are authorized by verifying the voter specified in `GovAction::VoteProposal`. Each vote key is tied to the correct validator and voter addresses, and the function ensures that only accounts in the `verifiers` set are permitted to participate in governance actions.

Proposal authorship and voter eligibility are validated through `GovAction::InitProposal` and `GovAction::VoteProposal`, respectively. Additionally, pre-storage checks in `is_valid_author` confirm that no proposal with the same ID and author already exists in the storage.

Votes are stored with a unique key structure: `[ADDRESS, "proposal", <proposal_id>, "vote", <validator_address>, <voter_address>]`, which ensures that each voter can cast only one vote per proposal. If a voter submits multiple votes, subsequent votes overwrite the initial one. Unauthorized votes are further prevented by signature verification, which ties transactions to the voter's address.

Scenarios where users attempt to create duplicate proposals, impersonate voters, or vote multiple times on the same proposal are all mitigated by these safeguards. Transactions from unauthorized users are rejected due to the governance VP's `verifiers` set checks.

Threat 1.2: Namada does not check for duplicate votes. Duplicate votes may be sent maliciously to exploit such a vulnerability or non-maliciously by a voter who does not see its vote processed after a while. 

Conclusion: The threat does not hold. Votes are stored using a unique key structure: `[ADDRESS, "proposal", <proposal_id>, "vote", <validator_address>, <voter_address>]`. This ensures that duplicate votes overwrite any existing vote for the same voter, proposal, and validator.

During tallying in `finalize_block`, votes are processed into runtime HashMaps for validators and delegators. These structures inherently prevent duplicates because `HashMap::insert` replaces any previous entry with the same key. This guarantees that only the latest vote is considered in the tally.

Voting power computation at the proposal's activation epoch further improves defense against double counting. Stake changes or redelegations during the voting period are resolved by considering only the bond state at the activation epoch, ensuring accurate voting power attribution.

Scenarios analyzed include voters submitting multiple votes for the same proposal, changing their vote during the voting period, or redelegating and voting again. These scenarios are consistently resolved due to the key structure, HashMap deduplication, and the final bond checks. While the storage may hold multiple votes in some cases, tallying logic ensures only valid, non-duplicate votes are considered.

Threat 1.3: Namada mixes the votes of different proposals. ✓

Conclusion: The threat does not hold. Votes are stored with a unique key structure tied to the `proposal_id`, ensuring strict separation between votes for different proposals. Each proposal establishes its own namespace for votes.

The `get_proposal_votes` function retrieves votes explicitly associated with a given `proposal_id`. This ensures that only votes relevant to the targeted proposal are considered during tallying. The `finalize_block` logic isolates proposals by processing them individually, fetching only those active in the current epoch using `load_proposals`.

Scenarios analyzed include delegators voting multiple times on the same proposal, votes being added to a different proposal, and proposals with overlapping activation epochs. These scenarios are consistently resolved by the `proposal_id` isolation in storage, targeted retrieval of votes, and scoped execution during `finalize_block`.

Threat 1.4: Namada counts invalid votes as valid, e.g., votes that were installed before or after the proposals' voting period. ✓

Conclusion: The threat does not hold in the current implementation. Votes are validated through the `is_valid_voting_window` function, which ensures that only votes cast within the designated start and end epochs of the voting period are considered. This safeguard effectively filters out votes that fall outside the allowed time frame.

Validator votes are further validated using the `is_active_validator` check, ensuring that only votes from active validators at the time of the proposal's activation epoch are included in the tally. Delegator votes tied to inactive or jailed validators are also excluded during this process, as their validity depends on the status of the associated validator.

Scenarios analyzed include votes cast outside the voting period, votes from validators who become inactive or jailed, and votes from reactivated validators.

Threat 1.5: Vote overwriting is not correctly implemented such that the last valid vote does not overwrite previous ones. ✓

Conclusion: The threat does not hold in the current governance module implementation. The `compute_proposal_result` function ensures that delegators' votes are correctly accounted for, even when they differ from their validator's votes.

The `validator_vote_is_same_side` logic compares delegator votes with their validator's vote. If they differ, the delegator's stake is subtracted from the validator's tally and added to the tally associated with the delegator's vote. This mechanism ensures that delegators' voting power is accurately reflected in the final tally.

The tallying process employs a runtime HashMap to dynamically adjust voting power, effectively managing situations where delegators vote differently from their validator.

Scenarios analyzed include delegators voting in alignment or opposition to their validator's vote and cases where delegators' votes differ among themselves. In all cases, the logic correctly adjusts the tallies to reflect the voting power of individual delegators, ensuring an accurate and fair tallying process.

Threat 1.6: The voting power computation is incorrect. ✓

Conclusion: The threat does not hold. The computation of voting power for both validators and delegators is scoped to the activation epoch, ensuring accurate and fair tallies.

Validator voting power is retrieved using the `PoS::read_validator_stake` function, which dynamically fetches the stake at the activation epoch. This guarantees that only valid and active voting power is included.

Delegator voting power is calculated using the `PoS::bond_amount` function. This function respects redelegations, unbonds, and stake changes, ensuring the tally reflects the state as of the activation epoch without including outdated or double-counted values.

All calculations are scoped to the activation epoch. This prevents discrepancies caused by stake adjustments or redelegations during the voting period. Additionally, inactive or jailed validators are excluded from the tally through the `is_active_validator` function, maintaining the integrity of the voting process.

Scenarios analyzed include changes to validator or delegator stakes, redelegations, and validator status changes during the voting period. In all cases, the tallying logic correctly reflects the voting power at the activation epoch, ensuring consistency and accuracy.

Threat 1.7: Validators can vote in the last third of the voting period. ✓

Conclusion: The threat does not hold. The governance VP enforces strict validation on the voting period for validators, ensuring that their votes are only considered if submitted within the first two-thirds of the voting period.

The function `is_valid_validator_voting_period`, called during vote validation, ensures compliance with the rule. It verifies that the current epoch is less than the start epoch plus two-thirds of the total voting period length (`end_epoch - start_epoch`). If this condition is not met, the validator's vote is rejected.

In `finalize_block`, when tallying votes during the activation epoch, only votes validated by the VP are included in the computation. This mechanism ensures that votes cast in the last third of the voting period are excluded from the tally.

2. An accepted governance proposal is executed exactly once at the beginning of its activation epoch

Threat 2.1: Malicious users can mutate proposals' activation epoch such that Namada executes them more than once. This assumes that Namada does not garbage collect them. ✓

Conclusion: The threat does not hold in the current implementation. The VP enforces validation to ensure that a pre-existing activation epoch key associated with a proposal ID cannot be overwritten. Transactions attempting to modify the activation epoch of an existing proposal fail validation.

Proposals are uniquely identified by their `proposal_id`, with the activation epoch and proposal ID tightly linked in storage via the proposal commit key. This ensures proposals are executed only when their activation epoch matches the current epoch.

During execution in `finalize_block`, proposals are fetched and processed exclusively based on their activation epoch. This mechanism prevents duplicate execution of the same proposal ID.

Scenarios analyzed include attempts to modify a proposal's activation epoch or reintroduce a proposal with the same `proposal_id` but a different activation epoch. Both scenarios are effectively blocked by VP validation and the underlying storage structure.

Threat 2.2: The logic to decide when an accepted proposal is executed is incorrect and leads to executing accepted proposals before or after its activation epoch. ✓

Conclusion: The threat does not hold in the current implementation. The `load_proposals` function, invoked during `finalize_block`, ensures that only proposals with an activation epoch matching the current epoch are considered for execution. This filtering guarantees that no proposals are executed prematurely or beyond their designated activation epoch.

Each proposal must have a valid committing key in storage, which links the proposal to its activation epoch. Proposals lacking this key are automatically excluded from execution. The function `force_read` further validates the activation epoch key, ensuring its correctness and presence in storage before execution.

Execution is handled through `execute_governance_proposals`, which processes only those proposals satisfying the condition `current_epoch == activation_epoch`.

Scenarios analyzed include proposals with mismatched activation epochs, which are filtered out during `load_proposals`, and proposals submitted with invalid or missing activation epochs, which are rejected during validation.

Threat 2.3: The logic to decide when an accepted proposal is executed is incorrect and leads to executing accepted proposals more than once. ✓

Conclusion: The threat does not hold in the current implementation. The `load_proposals` function ensures that each proposal is fetched exactly once per `finalize_block` call. This is enforced by the use of a BTreeSet, which inherently prevents duplicate entries in the set of proposal IDs. The function filters proposals by activation epoch, only including those matching the current epoch.

The governance VP validates the activation epoch key using `force_read`. Any attempt to tamper with the activation epoch or duplicate the committing key for an existing proposal is rejected.

Execution is handled iteratively in `execute_governance_proposals`. Proposals are processed from the unique set of IDs retrieved by `load_proposals`, with no mechanism to re-execute the same proposal within the same block.

Scenarios analyzed include valid proposals being fetched and executed once during their activation epoch, and malicious attempts to tamper with keys or activation epochs, which are prevented by validation checks in the governance VP.

Threat 2.4: The logic to decide when an accepted proposal is executed is non-deterministic. ✓

Conclusion: The threat does not hold in the current implementation. The `load_proposals` function filters proposals strictly by the current epoch, ensuring only proposals with an activation epoch matching the current epoch are included for execution. This eliminates any ambiguity in the selection process.

Activation epoch keys are validated using `force_read`, which confirms the existence and correctness of these keys in storage. Any proposals with incorrect or missing activation epoch keys are excluded, preventing inconsistencies.

Execution is deterministic due to the use of the BTreeSet structure, which enforces a consistent and predictable order for iterating over proposal IDs.

Scenarios analyzed include valid proposals with properly stored activation epochs being deterministically included and executed during their designated epoch, and proposals with incorrect or missing activation epoch keys being excluded from execution.

3. Governance proposal ids are unique

Threat 3.1: The governance VP accepts a proposal but does not check that the counter used to generate ids increases. ✓/✗

Conclusion: It is impossible if we assume that only whitelisted transactions can be executed because the `init_proposal` method called by the corresponding whitelisted transaction increases by one [here](#). When users submit arbitrary transactions, a transaction can install a new governance proposal without increasing the counter and compromise liveness; see [Malicious user could prevent honest users from installing valid governance proposals by not increasing the governance counter](#).

Threat 3.2: The governance VP accepts a proposal but does not check that a previously installed valid proposal has already used the proposal id. ✓/✗

Conclusion: It is impossible if we assume that only whitelisted transactions are executed. Pick an installed proposal at random. When this proposal was installed, the transaction read the current counter value `i` and increased it by one ([here](#)). Assume now that a new proposal is installed using `i` as the proposal id. This is impossible because to compute the proposal id, the transaction reads the counter's current value, which must be $\geq i+1$. This is under the assumption that no executed `DeafultWithWasm` governance proposal decreases the counter in between and deletes the proposal's associated keys.

Assume now that users can submit arbitrary transactions. Since the previously installed proposal is valid, then all its required keys are written to storage, including the `start_epoch` key. Assume that the new proposal also writes all required keys. This means that it will write `start_epoch` as well using the same key. Hence, [this](#) check in `is_valid_start_epoch` will make the governance VP reject the transaction, as required. Finally, assume that the new proposal only writes a subset of the keys. Then it is possible to craft a transaction that only writes for instance the type or the funds keys of the existing transaction and still have the governance VP to accept it; see [Proposal type key modification](#) and [Proposal funds key modification](#).

4. Installed governance proposals are immutable

Threat 4.1: The governance VP may accept a proposal that mutates any governance key associated with other proposals. ✓/✗

Conclusion: It is impossible when limited to whitelisted transactions only. However, a vulnerability exists when arbitrary transactions are allowed. Check the details in the associated findings, [Proposal type key modification](#) and [Proposal funds key modification](#).

5. The governance VP rejects invalid governance proposal transactions

Threat 5.1: The governance VP accepts governance proposal transactions that do not write all the required keys into storage. ✓

Conclusion: The check for `content`, `author`, `voting_start_epoch`, `voting_end_epoch`, and `activation_epoch` is done in the function `is_valid_init_proposal_key_set`. The check for `committing_epoch` is done in the `is_valid_activation_epoch` function. This function [will be called](#) because `is_valid_init_proposal_key_set` guarantees that the `activation_epoch` is among the keys.

Threat 5.2: The governance VP accepts governance proposal transactions that write any key associated with another proposal. ✓/✗

Conclusion: The property holds for whitelisted transactions. It does not hold for arbitrary transactions. Check the details in related findings, [Adding an incomplete proposal may cause panic in finalize_block](#) and [A governance proposal may prevent creation of future proposals](#).

Threat 5.3: The governance VP accepts governance proposal transactions that do not guarantee the conditions inferred from the governance parameters. This includes all the conditions in the proposal validity definition that involve any governance parameter. ✓

Conclusion: There are correct checks for `content size`, `minimal voting period`, `maximum latency`, `minimum grace period`, `max_proposal_period`, and minimum funds (split by cases, [here](#) and [here](#)).

Threat 5.4: The governance VP accepts governance proposal transactions with a start, end, or activation epoch less or equal to the current epoch. ✓

Conclusion: There is a check for the `start_epoch` and `end_epoch` being greater than the `current_epoch`. Furthermore, there is a `check` for the `activation_epoch` being greater than the `end_epoch`, implying the relation to the `current_epoch`, too.

Threat 5.5: The governance VP validates and accepts transactions with no governance action that modify governance keys. ✓/✗

Conclusion: Considering only whitelisted transactions, the property is preserved. If we consider arbitrary transaction, the property is violated by this threat, resulting in issues [A governance proposal may prevent creation of future proposals](#) or [Adding an incomplete proposal may cause panic in finalize_block](#).

Threat 5.6: The governance VP accepts governance proposal transaction of a type different than `DefaultWithWasm` that writes the `PROPOSAL_CODE` governance key. ✓

Conclusion: There is a `check` in `is_valid_proposal_code`.

Threat 5.7: The governance VP accepts a `DefaultWithWasm` governance proposal transaction that includes a WASM code of a size that exceeds the maximum allowed defined by `max_proposal_length`. ✓

Conclusion: There is a `check` in `is_valid_proposal_code`.

Threat 5.8: The governance VP accepts a `PGFSteward` governance proposal transaction that attempts to add more than one steward. ✓

Conclusion: There is a `check` in `is_valid_proposal_type`.

Threat 5.9: The governance VP accepts a `PGFSteward` governance proposal transaction that the proposed steward did not author.

Conclusion: There is a `check` in `is_valid_proposal_type`. ✓

6. The governance VP rejects invalid vote proposal transactions

Threat 6.1: The governance VP accepts votes that vote for a non-existing proposal. ✓


Conclusion: There is a `counter` variable that starts at `0`. For each new proposal that is added, the `counter` value is used as `proposal_id` and subsequently, the `counter` is increased by `1`. This implies that each vote that comes, must be a vote for a `proposal_id` that is strictly lower than the current value of the counter. This is checked in `is_valid_vote_key`.

Threat 6.2: The governance VP accepts votes for a given proposal outside the voting period. ✓

Conclusion: There is a `check` in `is_valid_voting_window`, called from `is_valid_vote_key`


Threat 6.3: The governance VP accepts votes from voters who are not validators or delegators. ✓

Conclusion: If the voter is a validator, the result of the check [is returned directly](#). Then, if it is not a delegator (and it is not a validator either, since this place in the code couldn't be reached if it were), an error is [returned](#).


Threat 6.4: The governance VP accepts votes from validators outside 2/3 of the voting period. 

Conclusion: The function `is_valid_validator_voting_period`, called from `is_valid_vote_key`, checks that `current_epoch < start_epoch + 2/3 * (end_epoch - start_epoch)`. We note a refactoring of the implementation may be necessary, as described in [Miscellaneous Code Findings](#).

7. The governance VP rejects invalid updates to governance parameters


Threat 7.1: The governance VP does not perform basic validity checks updates, e.g., `max_proposal_period` is greater than 0. 

Conclusion: Those checks are not performed and **this is by design** (proposal voters are expected to perform those checks themselves). In the finding [Some checks for governance-vetted updates are justified and beneficial](#), we argue that it may be a good idea to re-think that choice.

Threat 7.2: The governance VP does not check that a governance parameter can only be updated via an accepted governance proposal. 


Conclusion: There is [a check](#) before changing any governance parameter for whether the corresponding proposal has been accepted. It is worth emphasizing that this part of the implementation could be made more explicit, as we argued in [Some VP-functions are rendered trivial by their reachability conditions](#).

8. The pgf VP rejects invalid resign steward transactions


Threat 8.1: The pgf VP accepts resign steward transactions without guaranteeing that the steward signature is checked. 

Conclusion: The threat does not hold. When `validate_tx` is called, the steward is retrieved from the storage. In case the steward does not exist, the assumption is that the tx is removing the steward. Their signature is then checked, and if the `verifiers` do not contain `steward_address` an error is returned (code [ref](#)).


9. The pgf VP rejects invalid update steward commission transactions

Threat 9.1: The pgf VP accepts update steward commission transactions without guaranteeing that the steward signature is checked. 

Conclusion: The threat does not hold. When `validate_tx` is called, the steward is retrieved from the storage. If the steward exists, the assumption is that the tx is updating the reward distribution. Their signature is then checked, and if the `verifiers` do not contain `steward_address` an error is returned (code [ref](#)).

Threat 9.2: The pgf VP accepts update steward commission transactions that include negative reward percentages. 

Conclusion: The threat does not hold. When the function `is_valid_reward_distribution` is called, it verifies whether each percentage value in the `reward_distribution`. If the percentage is less than zero the function returns false (code [ref](#)). If the reward distribution is invalid, the error is returned (code [ref](#)).

Threat 9.3: The pgf VP accepts update steward commission transactions that include reward percentages greater than 100%. 

Conclusion: The threat does not hold. When the function `is_valid_reward_distribution` is invoked, it verifies whether each percentage value in the `reward_distribution`. If the percentage is greater than one the function returns false (code [ref](#)). If the reward distribution is invalid, the error is returned (code [ref](#)).

Threat 9.4: The pgf VP accepts update steward commission transactions that increase the number of reward destination addresses over the `REWARD_DISTRIBUTION_LIMIT` limit. ✓

Conclusion: The threat does not hold. When `is_valid_reward_distribution` is called upon the post steward details value, the size of the `reward_distribution` field within `StewardDetail` is checked. If the length of `reward_distribution` exceeds the `REWARD_DISTRIBUTION_LIMIT`, the function returns false (code ref). If the reward distribution is invalid, the error is returned (code ref).

Threat 9.5: The pgf VP accepts update steward commission transactions whose total reward percentage is greater than 100%. ✓

Conclusion: The threat does not hold. When the function `is_valid_reward_distribution` is invoked, it verifies whether the total percentage value exceeds one. If the total percentage is greater than one the function returns false (code ref). If the reward distribution is invalid, the error is returned (code ref).

10. The pgf VP rejects transactions that attempt to modify any protected pgf key

Threat 10.1: The pgf VP accepts transactions that modify the `KeyType::Fundings` key. ✓

Conclusion: The threat does not hold. When iterating through `changed_keys`, if the type of the modified key is `KeyType::Fundings`, an error will be returned (code ref).

Threat 10.2: The PGF VP accepts transactions that either lack data or modify a governance proposal that has not yet been accepted, thereon altering the `KeyType::PgfiInflationRate` key. ✓

Conclusion: The threat does not hold. When iterating through `changed_keys`, if the type of the modified key is `PgfiInflationRate`, the function `is_valid_parameter_change` will be called (code ref). If the tx data is empty or the accepted proposal is not being executed, an error is returned.

Threat 10.3: The PGF VP accepts transactions that either lack data or modify a governance proposal that has not yet been accepted, thereafter altering the `KeyType::StewardInflationRate` key. ✓

Conclusion: The threat does not hold. When iterating through `changed_keys`, if the type of the modified key is `StewardInflationRate`, the function `is_valid_parameter_change` will be called (code ref). If the tx data is empty or the accepted proposal is not being executed, an error is returned.

11. Assume a rejected `PGFPayment` proposal authored by a steward. The protocol removes the steward if the proposal receives more than 2/3 of `nay` votes over 2/3 of the total voting power at the proposal's activation epoch

Threat 11.1: The protocol does not remove a steward that authored a rejected `PGFPayment` proposal that received more than 2/3 of `nay` votes over 2/3 of the total voting power. ✓

Conclusion: The threat does not hold. The governance module ensures the proper removal of stewards when a `PGFPayment` proposal is rejected with more than 2/3 nay votes over 2/3 of the total voting power at the proposal's activation epoch. The `compute_proposal_result` function evaluates the voting results against the specified thresholds, and if the rejection criteria are met, the `remove_steward` function ensures the steward is removed from the stewards set.

For example, when a `PGFPayment` proposal authored by a steward is rejected with sufficient nay votes, the tally computation confirms the rejection, and the steward is immediately removed after the result is recorded. In the scenario where two `PGFPayment` proposals are authored by the same steward and the first proposal's rejection leads to the steward's removal, the second proposal is treated differently, as it is no longer authored by a steward. This results in different tallying rules being applied. This behavior was observed to function correctly during the analysis.

12. The protocol ensures that the number of stewards stays within the limit at all times

Threat 12.1: The protocol does not enforce correctly that the number of stewards stays within the limit when it executes an accepted `PGFSteward` proposal. ✓

Conclusion: The threat does not hold. The maximum number of stewards is enforced by obtaining the parameter at the beginning of the execution of `PGFSteward` proposals and verifying that the total number remains within the limit whenever a new steward is added.

For example, when a `PGFSteward` proposal to add a new steward is executed, the `finalize_block` function includes a check to confirm that the total number of stewards does not exceed the maximum parameter. Additionally, for a `PGFSteward` proposal to update a steward's commission that can implicitly involve adding a new steward, the validity predicate verifies that the steward is part of the verifiers set, ensuring the addition is legitimate. Both scenarios demonstrate that the protocol correctly enforces the maximum number of stewards while validating new additions.

13. The pgf inflation logic is correct and fair

Threat 13.1: `pgf_inflation_amount` is computed wrongly such that more tokens than expected are credited (minted) to the pgf account. Correctly means that it is computed as $(total_supply * pgf_inflation_rate) / epochs_per_year$. ✓

Conclusion: The threat does not hold. The `pgf_inflation_amount` is computed correctly, ensuring that the minted tokens align with the expected formula $(total_supply * pgf_inflation_rate) / epochs_per_year$.

The computation involves the `mul_floor` function, which multiplies the total supply by the inflation rate. This step explicitly checks for negative inflation rates, returning an error if any invalid value is provided. It also safeguards against overflow during multiplication, ensuring robust error handling.

The subsequent division by `epochs_per_year` is handled by the `checked_div_u64` function, which avoids division by zero by returning `None` when the divisor is zero. In such cases, the calculation fails gracefully, preventing any unintended inflation.

Precision is managed through rounding down in `mul_floor`, ensuring that no more tokens than intended are minted. While this introduces minor precision loss, it ensures the system does not exceed the defined inflation limits. These safeguards collectively ensure that the inflation amount remains accurate and within protocol expectations.

Threat 13.2: The iteration over continuous funding is non-deterministic. ✓

Conclusion: The threat does not hold. The iteration over continuous fundings is deterministic due to the sorting step implemented in the `get_continuous_pgf_payments` function. Specifically, the code retrieves all

continuous fundings and explicitly sorts them by the proposal ID before iterating. This ensures a consistent order of processing across different executions.

The `pgf_fundings.sort_by(|a, b| a.id.cmp(&b.id))` line prioritizes payments by the oldest governance proposal ID, ensuring that the iteration order is stable and repeatable, regardless of the underlying storage order.

The `get_continuous_pgf_payments` function collects all funding data into a `Vec<StoragePgffunding>`, handling any errors during the retrieval process gracefully by filtering out invalid data.

Threat 13.3: There is a bug in the addresses used to pay either steward rewards or continuous funds. ✓

Conclusion: The threat does not hold. The implementation ensures that the addresses used for steward rewards or continuous funds are validated for correctness. The governance module verifies that source and destination addresses align with the expected logic and cannot be arbitrarily substituted or manipulated. Checks ensure that addresses correspond to valid and authorized entities for the given operation and are directly linked to the proposal parameters, failing validation checks in the VP or execution logic if altered. These safeguards maintain the integrity of the reward and funding mechanisms.

Threat 13.4: The total amount of funds transferred due to continuous fund transfers exceeds `pgf_inflation_amount`. ✓

Conclusion: The threat does not hold in the current implementation. The PGF account is credited with the calculated `pgf_inflation_amount` during `apply_inflation`, ensuring that this amount serves as the upper limit for all continuous payments. Continuous payments are processed sequentially using the `TransToken::transfer` function, and if the PGF account is insufficient to cover all transfers, payments beyond the available funds fail gracefully without exceeding the allocated amount. The `pgf_fundings` list is deterministically prioritized by governance proposal id, and only valid entries retrieved from storage are processed, preventing unauthorized additions. **Note:** If the funds are fully drained during distribution, subsequent users receive no funds, with only an error indicating the insufficiency. The Namada team has clarified that handling such cases is a responsibility delegated to validators.

Threat 13.5: If there are not enough funds to make all transfers, the algorithm is fair with all projects. ✓

Conclusion: The threat does not hold in the current implementation. The continuous funding mechanism processes payments in a deterministic order, prioritizing projects based on the oldest governance proposal id first. This ensures fairness in how funds are allocated when the PGF account does not have sufficient balance to cover all transfers. Projects with older proposal ids are paid first, and payments fail gracefully for projects later in the list if the account runs out of funds. **Note:** If the funds are fully drained during distribution, the remaining projects receive no payment, and only an error indicates the issue. According to the Namada team, addressing such scenarios is expected to be handled by the validators.

Threat 13.6: `pgf_steward_inflation` is computed wrongly such that more tokens than expected are credited (minted) per steward. Correctly means that it is computed as `(total_supply*stewards_inflation_rate)/epochs_per_year`. ✓



Conclusion: The threat does not hold. The computation of `pgf_steward_inflation` is correctly implemented as `(total_supply * stewards_inflation_rate) / epochs_per_year`. The `mul_floor` function is used to multiply the total supply by the inflation rate, ensuring the result is rounded down to avoid any overflow or excess allocation. This amount is then divided by the number of epochs per year using `checked_div_u64`, which prevents division by zero and gracefully handles any potential overflow.

Threat 13.7: Reward distribution is done wrongly such that the total amount of distributed rewards exceeds `pgf_steward_inflation`. ✓

Conclusion: The threat does not hold. The reward distribution for stewards is implemented such that the total amount distributed cannot exceed the calculated `pgf_steward_inflation`. Each steward's reward is determined by multiplying the inflation amount by their specified percentage, using the `mul_floor` function, which rounds down to ensure no excess allocation. Additionally, the percentages for reward distribution are [validated during their definition](#), ensuring they do not collectively exceed 100%. A minor [code improvement](#) is made from this threat regarding redundant error handling.

Liveness Properties


14. Given a valid governance proposal transaction, the governance VP accepts it

Threat 14.1: The governance VP rejects a governance proposal transaction that is valid according to the validity definition in [System Overview](#).  

Conclusion: There is a check for the `max_proposal_period` parameter, which checks for the number of epochs [between the start and end of voting](#), while it should be checking for the number of epochs between the start of voting and the activation of the proposal, as described in [A wrong check for max_proposal_period](#).


There is a check requiring that `start_epoch > current_epoch`. We flag this in [Voting starting epoch starts only strictly after the proposal epoch](#), as being unclear whether this was the intended validity predicate condition, or rather `start_epoch >= current_epoch`.

If we are considering arbitrary transactions, additional problems may occur, as described in [A governance proposal may prevent creation of future proposals](#) and [Malicious user could prevent honest users from installing valid governance proposals by not increasing the governance counter](#)

Threat 14.2: The governance VP rejects a governance proposal transaction created legally because its proposal id is already in use. 

Conclusion: In the context of using only whitelisted transactions, this cannot happen since the system generates the sequence of proposal ids.

15. Given a valid vote transaction, the governance VP accepts it

Threat 15.1: The governance VP rejects a vote transaction that is valid according to the vote validity definition in [System Overview](#). 

Conclusion: The function `is_valid_vote_key` contains the following checks:

1. Current counter is lower than or equal to the `proposal_id` of the voted proposal (the proposal exists)
2. The vote happens in between `vote_start_epoch` and `vote_end_epoch`
3. If the voter is validator, the vote happens within the first 2/3 of the allowed voting period
4. If not validator, the voter is a delegator.

These checks ensure the property, given that a validator cannot delegate its funds to other validators. It is worth mentioning that a validator, even with its own self-delegated funds, can only vote in the first 2/3 of the allowed voting period (but this is a valid design choice).

16. Assume a governance proposal that has enough valid votes to be accepted by the beginning of its activation epoch, then Namada executes it

Threat 16.1: The voting power computation is incorrect. 

Conclusion: The threat of incorrect voting power computation is mitigated in Namada's governance module. The `compute_proposal_votes` function dynamically aggregates votes and calculates voting power based on the

state at the activation epoch. Validator voting power is retrieved using `PoS::read_validator_stake`, and delegator voting power is computed through `PoS::bond_amount`. Both functions ensure that the final tally reflects the accurate delegation and stake state at the activation epoch.

Duplicate votes from the same validator or delegator are handled by the HashMap structure in `compute_proposal_votes`, ensuring only the most recent vote is counted. Votes from inactive or jailed validators are excluded using `PoS::is_active_validator`. Changes in delegation or stake during the voting period are reconciled at the activation epoch, preventing overcounting or undercounting and ensuring the liveness of the governance process.

Threat 16.2: The computation on when to check the acceptance of a governance proposal is incorrect. ✓

Conclusion: The threat does not hold. The governance module implements a deterministic process for checking proposal acceptance. Proposals are retrieved using the `load_proposals` function, which ensures only proposals with an activation epoch matching the current epoch are considered. The governance VP validates consistency between the activation epoch and commit key, preventing any mismatched or tampered proposals from being processed. Proposals missing a valid commit key are excluded, ensuring only correctly stored proposals are fetched and executed.

Threat 16.3: The set of votes considered when computing acceptance excludes some valid votes or includes some invalid ones that determine the result. ✓

Conclusion: The governance module accurately includes valid votes and excludes invalid or duplicate votes through the `compute_proposal_votes` function. Votes are tied to a specific proposal ID, validator, and delegator using a unique key structure that prevents overlaps. The use of a HashMap ensures that only the most recent vote from each delegator-validator pair is counted.

Votes from jailed or inactive validators are excluded by `PoS::is_active_validator`, while delegator votes are only included if the delegation is valid and active at the activation epoch, verified through `PoS::bond_amount`. This approach prevents both overcounting and undercounting of votes.

While changes in stake or delegation during the voting period are reconciled at the activation epoch, this introduces minor inconsistencies where the tallying results reflect the state at the activation epoch rather than the moment of voting. Despite this, the computation ensures that the final result is based on accurate and valid data as of the activation epoch, maintaining the correctness of the process.

17. Given a valid update steward commission transaction transaction, the pgf VP accepts it

Threat 17.1: The pgf VP rejects a update steward commission transaction that it is valid according to the update steward commission validity definition in [System Overview](#). ✓

Conclusion: The threat does not hold. The `validate_tx` function performs several checks:

- It verifies that `total_stewards_pre` is less than `total_stewards_post`.
- It checks if the `get_steward` function successfully returns a steward.
- It conducts a signature validation.

An interesting scenario would be if a user submits an arbitrary transaction in which the steward resigns and updates the reward distribution. In this case, while iterating through `changed_keys` in `validate_tx` (code [ref](#)), the `get_steward` function would fail to return a steward. Consequently, the check for `is_valid_reward_distribution` would be bypassed, leading to a change in reward distribution to potentially invalid values. However, this scenario is impossible. Even with arbitrary transactions, the

`update_commission` function ensures that the steward is inserted alongside the `StewardDetail`. If the steward has been previously removed or does not exist, it will be created.

Threat 17.2: The pgf VP rejects a update steward commission transaction that can be instantiated via the `update_steward_commission` predefined transaction. ✓

Conclusion: The threat does not hold. The predefined transaction `update_steward_commission` inserts the steward address into verifiers and calls the function `update_steward_commission`, where the steward commission is updated. If the `steward_commission` data is valid the transaction will pass the VP check.

18. Given a valid resign steward transaction transaction, the pgf VP accepts it

Threat 18.1: The pgf VP rejects a resign steward transaction that it is valid according to the resign steward validity definition in [System Overview](#). ✓

Conclusion: Similarly to threat 17.1, the threat does not hold.

The `validate_tx` function executes several checks:

- It ensures that `total_stewards_pre` is less than `total_stewards_post`.
- It verifies whether the `get_steward` function successfully retrieves a steward. If this retrieval is successful, the reward distribution is validated using the `is_valid_reward_distribution` function.
- It performs a signature validation.

Threat 18.2: The pgf VP rejects a update steward commission transaction that can be instantiated via the `remove_steward` predefined transaction. ✓

Conclusion: The threat does not hold. The predefined transaction `remove_steward` inserts the steward address into verifiers and calls the function `remove_steward`, where the steward is removed from `storage`. If no error is returned, the transaction will pass the VP check.

Findings

Title	Scope	Type	Severity	Status
Malicious user could prevent honest users from installing valid governance proposals by not increasing the governance counter	Governance and PGF	IMPLEMENTATION	With whitelisted transactions: 0 NONE With arbitrary transactions: 4 CRITICAL	ACKNOWLEDGED
A governance proposal may prevent creation of future proposals	Governance and PGF	IMPLEMENTATION	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL	ACKNOWLEDGED
Adding an incomplete proposal may cause panic in <code>finalize_block</code>	Governance and PGF	IMPLEMENTATION	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL	ACKNOWLEDGED
Proposal type key modification	Governance and PGF	IMPLEMENTATION	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL	ACKNOWLEDGED
Proposal funds key modification	Governance and PGF	IMPLEMENTATION	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL	ACKNOWLEDGED
Some checks for governance-vetted updates are justified and beneficial	Governance and PGF	PROTOCOL	2 MEDIUM	ACKNOWLEDGED

Title	Scope	Type	Severity	Status
Voting starting epoch starts only strictly after the proposal epoch	Governance and PGF	LOW	LOW	ACKNOWLEDGED
There's no need to deal with multiple governance proposals being installed in parallel	Governance and PGF	IMPLEMENTATION PROTOCOL	0 INFORMATIONAL	ACKNOWLEDGED
A wrong check for max_proposal_period	Governance and PGF	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Max Proposal Latency Parameter Not Included in the is_parameter_key function	Governance and PGF	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Some VP-functions are rendered trivial by their reachability conditions	Governance and PGF	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Miscellaneous Code Findings	Governance and PGF	IMPLEMENTATION PRACTICE	0 INFORMATIONAL	ACKNOWLEDGED

Malicious user could prevent honest users from installing valid governance proposals by not increasing the governance counter

Project	Namada Governance and PGF
Type	IMPLEMENTATION
Severity	With whitelisted transactions: 0 NONE With arbitrary transactions: 4 CRITICAL
Impact	3 HIGH
Exploitability	With whitelisted transactions: 0 NONE With arbitrary transactions: 3 HIGH
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [crates/governance/src/vp/mod.rs](#)

Description

Namada maintains a counter in state that it is used to compute unique governance proposals' ids. The governance vp assumes that transactions that attempt to install new governance proposals use the current value of the counter to compute the proposal id and then increase it. Nevertheless, this is not enforced by the governance vp, and one may install a governance proposal without increasing the counter. If this occurs, successive valid transactions that attempt to install governance proposals will be rejected by the vp, compromising liveness.

Problem Scenarios

- Assume that a malicious user submits a transaction that writes some of the required keys, e.g., `start_epoch`, of a new governance proposal with the proposal id being generated from the current counter value.
- Assume that the transaction does not increase the counter, i.e., `post_counter==pre_counter`, nor writes the proposal commit key.
- The `is_valid_init_proposal_key_set` function will then return `(true, 0)` without checking for mandatory keys. Note that in the function, even if it does, the proposal commit key is not part of the mandatory keys.
- Since it does not write the proposal commit key, the vp does not call `is_valid_proposal_commit` isn't called, which is the only place in the vp where the non-increasing of the counter would have been detected. Thus, the transaction is accepted by the governance vp.

- Assume now that a new transaction submitted by an honest user attempts to install new governance proposal. This will be rejected as the proposal id will clash with the proposal submitted by the malicious user, e.g., the checks in `is_valid_start_epoch` won't pass.

Recommendation

We recommend adding an explicit check in the governance vp that verifies that the counter is increased if a new proposal is installed. Note that simply strengthening [this condition](#) won't work as this will prevent vote transactions from being accepted. One can only do the check on the counter increase if the transaction is installing a new proposal.

A governance proposal may prevent creation of future proposals

Project	Nadama Governance and PGF
Type	IMPLEMENTATION
Severity	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL
Impact	3 HIGH
Exploitability	With whitelisted transactions: 0 NONE With arbitrary transactions: 3 HIGH
Status	ACKNOWLEDGED
Issue	

This finding resembles in its effect the one described in [Malicious user could prevent honest users from installing valid governance proposals by not increasing the governance counter](#). However, the mechanics (and, thus, potential addressing) of causing the problem differ.

Involved artifacts

- [namada/crates/governance/src/vp/mod.rs](#)

Description

A governance proposal that modifies all the required keys for the governance proposal for the proposal id `a`, and additionally some keys needed for the proposal id `a+1` (but not the counter) will pass the validity predicate. The reason is that [the check for mandatory keys](#) will be satisfied for `a`, and won't be checked for `a+1` since the counter was not changed.

Consider the following snippet illustrating the problem:

```
fn test_governance_proposal_writing_unrelated() {
    let mut state = init_storage();
    let activation_epoch = 19;
    let keys_changed_0 = get_proposal_keys(0, activation_epoch);
    let gas_meter = RefCell::new(VpGasMeter::new_from_tx_meter(
        &TxGasMeter::new(u64::MAX),
    ));
    let (vp_wasm_cache, _vp_cache_dir) =
        wasm::compilation_cache::common::testing::vp_cache();

    let tx_index = TxIndex::default();
```

```

let signer = keypair_1();
let signer_address = Address::from(&signer.clone().ref_to());
let verifiers = BTreeSet::from([signer_address.clone()]);

initialize_account_balance(
    &mut state,
    &signer_address.clone(),
    token::Amount::native_whole(5010),
);
initialize_account_balance(
    &mut state,
    &ADDRESS,
    token::Amount::native_whole(0),
);
state.commit_block().unwrap();

let tx_code = vec![];
let tx_data = vec![];

let mut tx = Tx::from_type(TxType::Raw);
tx.header.chain_id = state.in_mem().chain_id.clone();
tx.set_code(Code::new(tx_code, None));
tx.set_data(Data::new(tx_data));
tx.add_section(Section::Authorization(Authorization::new(
    vec![tx.header_hash()],
    [(0, keypair_1())].into_iter().collect(),
    None,
)));

// first, init_proposal for proposal_id 0
init_proposal(
    &mut state,
    0,
    500,
    3,
    9,
    19,
    &signer_address,
    false,
);

let batched_tx = tx.batch_ref_first_tx().unwrap();
let ctx = Ctx::new(
    &ADDRESS,
    &state,
    batched_tx.tx,
    batched_tx.cmt,
    &tx_index,
    &gas_meter,
    &keys_changed_0,
    &verifiers,
    vp_wasm_cache.clone(),
);

// this should return true because state has been stored
assert_matches!(

```

```

    GovernanceVp::validate_tx(
        &ctx,
        &batched_tx,
        &keys_changed_0,
        &verifiers
    ),
    Ok(_)
);

state.write_log_mut().commit_batch_and_current_tx();
state.commit_block().unwrap();

// then, create a proposal for id=1
let keys_changed_1 = get_proposal_keys(1, activation_epoch);
init_proposal(
    &mut state,
    1,
    500,
    3,
    9,
    19,
    &signer_address,
    false,
);

// Now, add some extra keys relating to id=2
let voting_end_epoch_key_2 = get_voting_end_epoch_key(2);
let _ = state
    .write_log_mut()
    .write(&voting_end_epoch_key_2, Epoch(21).serialize_to_vec())
    .unwrap();

let voting_start_epoch_key_2 = get_voting_start_epoch_key(2);
let _ = state
    .write_log_mut()
    .write(&voting_start_epoch_key_2, Epoch(17).serialize_to_vec())
    .unwrap();

// all_keys_changed is a union of keys_changed and the set containing
voting_end_epoch_key_2
let mut all_keys_changed = keys_changed_1.clone();
all_keys_changed.insert(voting_end_epoch_key_2);
all_keys_changed.insert(voting_start_epoch_key_2);

let ctx = Ctx::new(
    &ADDRESS,
    &state,
    batched_tx.tx,
    batched_tx.cmt,
    &tx_index,
    &gas_meter,
    &all_keys_changed,
    &verifiers,
    vp_wasm_cache.clone(),
);
dbg!(&all_keys_changed);

```

```

    // this returns true (even though it shouldn't since it wrote to an unrelated
    material)
    assert_matches!(
        GovernanceVp::validate_tx(
            &ctx,
            &batched_tx,
            &all_keys_changed,
            &verifiers
        ),
        Ok(_)
    );
}

```

Problem Scenarios

The described behavior allows us to write into `voting_start_epoch_key` and `voting_end_epoch_key` for `a+1`. When a future governance proposal comes, the validity predicate will necessarily `fail` the validity predicate, since it already has those keys written in the store, for instance, here:

```

let has_pre_start_epoch = ctx.has_key_pre(&start_epoch_key)?;
if has_pre_start_epoch {
    let error = Error::new_alloc(format!(
        "Failed to validate start epoch. Proposal with id \
        {proposal_id} already had a pre_start epoch written to \
        storage in its slot.",
    ));
    tracing::info!("{error}");
    return Err(error);
}

```

Recommendation

Consider restricting transactions that touch governance proposal keys onto those dealing only with one `proposal_id`.

Adding an incomplete proposal may cause panic in finalize_block

Project	Nadama Governance and PGF
Type	IMPLEMENTATION
Severity	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL
Impact	3 HIGH
Exploitability	With whitelisted transactions: 0 NONE With arbitrary transactions: 3 HIGH
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [namada/crates/governance/src/finalize_block.rs](#)
- [namada/crates/governance/src/vp/mod.rs](#)

Description

Incomplete proposals may be written, circumventing the checks for all the necessary keys. For instance, we could write a `committing_key`, unrelated to other parameters defining a governance proposal, and it will satisfy validity predicates.

```
let mut state = init_storage();

let activation_epoch = 19;
let keys_changed_0 = get_proposal_keys(0, activation_epoch);

let gas_meter = RefCell::new(VpGasMeter::new_from_tx_meter(
    &TxGasMeter::new(u64::MAX),
));
let (vp_wasm_cache, _vp_cache_dir) =
    wasm::compilation_cache::common::testing::vp_cache();

let tx_index = TxIndex::default();

let signer = keypair_1();
let signer_address = Address::from(&signer.clone().ref_to());
let verifiers = BTreeSet::from([signer_address.clone()]);
```

```

initialize_account_balance(
    &mut state,
    &signer_address.clone(),
    token::Amount::native_whole(5010),
);
initialize_account_balance(
    &mut state,
    &ADDRESS,
    token::Amount::native_whole(0),
);
state.commit_block().unwrap();

let tx_code = vec![];
let tx_data = vec![];

let mut tx = Tx::from_type(TxType::Raw);
tx.header.chain_id = state.in_mem().chain_id.clone();
tx.set_code(Code::new(tx_code, None));
tx.set_data(Data::new(tx_data));
tx.add_section(Section::Authorization(Authorization::new(
    vec![tx.header_hash()],
    [(0, keypair_1())].into_iter().collect(),
    None,
)));

// first, init_proposal for proposal_id 0
init_proposal(
    &mut state,
    0,
    500,
    3,
    9,
    19,
    &signer_address,
    false,
);

// create a committing key related to proposal_id = 1
let committing_key =
    get_committing_proposals_key(1, activation_epoch);

// Now, write the extra committing key
let _ = state
    .write_log_mut()
    .write(&committing_key, ().serialize_to_vec())
    .unwrap();

// all_keys_changed is a union of keys_changed and the set containing
// voting_end_epoch_key_2
let mut all_keys_changed = keys_changed_0.clone();
all_keys_changed.insert(committing_key);
// all_keys_changed.insert(voting_start_epoch_key_2);

let batched_tx = tx.batch_ref_first_tx().unwrap();
let ctx = Ctx::new(
    &ADDRESS,

```

```
    &state,  
    batched_tx.tx,  
    batched_tx.cmt,  
    &tx_index,  
    &gas_meter,  
    &all_keys_changed,  
    &verifiers,  
    vp_wasm_cache.clone(),  
);  
  
assert_matches!(  
    GovernanceVp::validate_tx(  
        &ctx,  
        &batched_tx,  
        &all_keys_changed,  
        &verifiers  
    ),  
    Ok(_)  
); \
```

Problem Scenarios

When we enter the activation epoch of the inserted `committing_key`, governance proposals that need to be executed are `loaded`. Among others, the inserted `committing_key` is loaded, and given as an argument `proposal_ids` to `execute_governance_proposals`. There, the funds under related `proposal_funds_key` are `force read`. Since they do not exist, this results in a panic and halts the chain.

Recommendation

This problem stems from allowing partial keys, and is related to [A governance proposal may prevent creation of future proposals](#). Like there, we recommend enforcing in the validity predicate that all the mandatory keys are present (which already is the case), and that all gov-related keys are related to a single `proposal_id`.

Proposal type key modification

Project	Namada Governance and PGF
Type	IMPLEMENTATION
Severity	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL
Impact	3 HIGH
Exploitability	With whitelisted transactions: 0 NONE With arbitrary transactions: 3 HIGH
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [crates/node/src/protocol.rs](#)
- [crates/governance/src/vp/mod.rs](#)
- [crates/governance/src/finalize_block.rs](#)

Description

When `check_vps` is called, `keys_changed` are retrieved from `write_log` (code ref). The function will include all changed keys in the `changed_keys` regardless of their type. The vps will then be executed by calling the function `execute_vps` (code ref) where `GovernanceVp::validate_tx` is called (code ref) in case any of the changed keys belong to the governance.

When assessing the keys associated with a proposal, the `changed_keys` are initially validated in the vp using the function `is_valid_init_proposal_key_set`. The code expects that if the transaction is a new governance proposal, the counter is increased. During the iteration through the values between `pre_counter` and `post_counter`, the `mandatory_keys` set will be constructed only for the proposal whose id is equal to the counter value and this set will be a subset of `changed_keys`.

When the `changed_keys` are later examined in the `validate_tx` function (code ref), they are validated against the `proposal_id` associated with each key.

Unless the prior existence of the key in question is verified - and an error is returned due to an attempt to modify its value - the value may be altered. The checks for previously existing values are missing in functions `is_valid_proposal_type` and `is_valid_funds`.

Problem Scenarios

- Assume that there exists a governance proposal with proposal id `id` in storage with all required fields.
- Assume now that a user submits an arbitrary transaction where it attempts to write a valid new proposal and also overwrite the proposal type key associated with the `id`.
- The function `is_valid_init_proposal_key_set` will accept this transaction.
- Since there is no check in `is_valid_proposal_type` whether the keys previously existed, the governance vp will consider the transaction valid.
- Given the above scenario, the proposal type could be changed from `PGFPayment` to `DefaultWithWasm` using an arbitrary transaction. This adjustment will impact the voting policy when `execute_governance_proposals` is called, as the `tally_type` is directly linked to the proposal type (code [ref](#)), and thus alter the number of votes required to pass an ongoing proposal. A malicious user could make the required number of votes higher and therefore postpone or change the outcome of the proposal.
- Additionally, if the type of proposal is changed from any type to `DefaultWithWasm`, the `is_valid_proposal_code` will only check wasm length, if the code previously existed, and if it's valid **post** proposal type (code [ref](#)). Consequently, this means that malicious users could add new wasm code to the ongoing proposal. In case the proposal passes, the injected code will be executed.

Recommendation

Add the check whether the `proposal_type` was previously written in the storage.

```
let has_pre_proposal_type =
  ctx.has_key_pre(&proposal_type_key)?;
if has_pre_proposal_type {
  return Err(Error::new_alloc(format!(
    "Proposal with id {proposal_id} already had proposal type \
      written to storage in its slot.",
  )));
}
```

Proposal funds key modification

Project	Namada Governance and PGF
Type	IMPLEMENTATION
Severity	With whitelisted transactions: 0 INFORMATIONAL With arbitrary transactions: 4 CRITICAL
Impact	3 HIGH
Exploitability	With whitelisted transactions: 0 NONE With arbitrary transactions: 3 HIGH
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [crates/node/src/protocol.rs](#)
- [crates/governance/src/vp/mod.rs](#)
- [crates/governance/src/finalize_block.rs](#)

Description

Similarly to what was outlined in the document [here](#), the value of the funds can be modified unless the prior existence of the funds key is verified. An error should be returned if there is an attempt to modify the value associated with the funds key. This step is missing in the function `is_valid_funds`.

Problem Scenarios

- Assume that there exists a governance proposal with proposal id `id` in storage with all required fields.
- Assume now that a user submits an arbitrary transaction where it attempts to write a valid new proposal and also overwrite the funds key associated with the `id`.
- The function `is_valid_init_proposal_key_set` will accept this transaction.
- Since there is no check in `is_valid_funds` whether the keys previously existed, the governance vp will consider the transaction valid.
- Based on the above scenario, the `funds_key` could be modified which will result in a change of funds that are used later in the `execute_governance_proposals` function (code [ref](#)).
- This adjustment affects the transfer and burning of funds based on the outcome of the proposal. A malicious user can raise the amount of funds that he receives. In case the proposal passes, the user could have modified the funds parameter to receive more than what was originally transferred to the governance internal address.

Recommendation

Add the check whether the funds were previously written in the storage.

```
let has_pre_funds =  
  ctx.has_key_pre(&funds_key)?;  
if has_pre_funds {  
  return Err(Error::new_alloc(format!(  
    "Proposal with id {proposal_id} already had funds \  
    written to storage in its slot.",  
  )));  
}
```

Some checks for governance-vetted updates are justified and beneficial

Project	Namada Governance and PGF
Type	PROTOCOL
Severity	2 MEDIUM
Impact	3 HIGH
Exploitability	1 LOW
Status	ACKNOWLEDGED
Issue	

Description

The design choice of the governance and PGF codebase is that a governance decision has the highest priority, and should not be restrained by validity predicates. The rationale behind it is that it is extremely difficult to navigate the tradeoff between the enforced checks and allowing governance enough freedom to update the codebase. Giving more freedom to governance avoids potential hard-forks.

While this rationale makes sense, we argue that there are situations in which restrictions should be placed risk-free, ensuring that some unwanted scenarios do not happen.

Problem Scenario

As a case-study, let's consider the parameter `max_proposal_period`, which is an upper limit on the number of epochs between the start epoch and the activation epoch. The validity predicate requires that `start_epoch < activation_epoch <= start_epoch + max_proposal_period`. It is clear that the requirement can possibly be satisfied only if `max_proposal_period > 0`. Thus, if `max_proposal_period` were changed into 0, no governance proposal could be passed (including the one meant to rectify the problem).

We believe that, whenever possible, automatic checks should be put in place to reduce the verification burden from voters.

Recommendation

We suggest re-evaluating the feasibility of introducing basic checks for all `governance` and `pgf` parameters (and also outside of this scope, if applicable). Some candidate parameters are: `max_proposal_code_size`, `max_proposal_content_size`, `max_proposal_latency`, `pgf_inflation_rate`, `stewards_inflation_rate`.

Voting starting epoch starts only strictly after the proposal epoch

Project	Namada Governance and PGF modules
Type	IMPLEMENTATION PRACTICE
Severity	INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [crates/governance/src/vp/mod.rs](#)

Description

When validating a proposal's `start_epoch`, the `is_valid_start_epoch` will [return an error](#) if `start_epoch <= current_epoch`. This way, it is not allowed for the voting to start in the current epoch already (`start_epoch == current_epoch`).

Problem Scenarios

This finding does not entail any security problem. The only problem it may create is unclear expectations by users. As clarified by the development team, this is a design choice. We report it here nonetheless to emphasise a potential wrong expectation.

There's no need to deal with multiple governance proposals being installed in parallel

Project	Namada Governance and PGF
Type	IMPLEMENTATION PROTOCOL
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [crates/governance/src/vp/mod.rs](#)

Description

Some parts of the governance vp assume that proposal ids are generated statically: when a user creates the transaction. In reality, for the predefined `init_proposal` transaction, proposal ids are generated dynamically when the associated transaction is executed by reading the current value of the governance counter. If there is no intent to support arbitrary transactions that write multiple governance proposals at once in the short term, we recommend that the governance vp works under the assumption that each transaction attempts to install at most one governance proposal. Furthermore, it should assume that its proposal id is generated from the current value of the governance counter.

Problem Scenarios

There are no problematic scenarios, but the code related to parallel proposals may be confusing for someone onboarding on the code, and a source of bugs or security concerns that can be avoided.

Recommendation

Assuming that there is no intention to allow transactions to install more than one governance proposal in a single transaction, we recommend removing all the code associated with tolerating the installation of multiple governance proposals from the governance vp. For instance, the code in `is_valid_init_proposal_key_set` should be modified to reject transactions that write more than one proposal ids. Also, the `set_count` variable won't be needed as it will always be equal to 1.

A wrong check for max_proposal_period

Project	Namada Governance and PGF
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [namada/crates/governance/src/vp/mod.rs](#)

Description

The validity predicate, as defined in `is_valid_end_epoch` checks that `end_epoch - start_epoch <= max_period`. The documentation, however, defines that it should be `activation_epoch - start_epoch <= max_period`. This correct check is indeed implemented in `is_valid_activation_epoch`.

Problem Scenarios

This mistake does not cause any safety violations of the validity predicates, since the correct check is implemented. Thus, the safety is guaranteed.

However, the current code may reject a submission that is valid according to the validity definition from the documentation because of the overly strict requirement `end_epoch - start_epoch <= max_period`. Furthermore, using the same parameter in two semantically different checks makes the code confusing and thus more difficult to maintain.

Recommendation

Remove the check for the maximum difference between the `end_epoch` and `start_epoch`, or use a different parameter if this needs to be enforced. More generally, as suggested in [Miscellaneous Code Findings](#), it makes sense to merge functions `is_valid_activation_epoch`, `is_valid_start_epoch`, and `is_valid_end_epoch`.

Max Proposal Latency Parameter Not Included in the `is_parameter_key` function

Project	Namada Governance and PGF
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [namada/crates/governance/src/vp/mod.rs](#)

Description

The predicate `is_parameter_key` is implemented in the following way

```
pub fn is_parameter_key(key: &Key) -> bool {
    is_min_proposal_fund_key(key)
    || is_max_content_size_key(key)
    || is_max_proposal_code_size_key(key)
    || is_min_proposal_voting_period_key(key)
    || is_max_proposal_period_key(key)
    || is_min_grace_epochs_key(key)
}
```

Notably, it returns `false` for the `max_proposal_latency_key`.

Problem Scenarios

The implementation of `is_parameter_key` will cause the `iteration over changed keys` in the validity predicate not match on `max_proposal_latency_key`. Instead, it will be matched by `KeyType::UNKNOWN_GOVERNANCE`, resulting in an error.

Because of the discussion in [Some VP-functions are rendered trivial by their reachability conditions](#), the `is_valid_parameter` check will anyway always going to result in an error. Thus, this bug does not cause a change in the behavior.

Nonetheless, the function should be implemented correctly so that one can rely on it for the future usage.

Recommendation

We suggest fixing the `is_parameter_key` function by adding into the disjunction the check for the `max_proposal_latency` key.

Some VP-functions are rendered trivial by their reachability conditions

Project	Namada Governance and PGF
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [namada/crates/governance/src/vp/mod.rs](#)
- [namada/crates/governances/src/vp/pgf.rs](#)
- [namada/crates/trans_token/src/vp.rs](#)

Description

Let's consider the `gov`'s validity predicate, and the call to function `is_valid_parameter`. This function returns `true` if it is called in the context of the accepted governance proposal (in other words, if `is_proposal_accepted` is true), and `false` otherwise.

When can `is_valid_parameter` be called at all? If `is_proposal_accepted` is `true`, the validity predicate will return prematurely, thus never reaching the call to `is_valid_parameter`. In such a setup, the logic of `is_valid_parameter` reduces to trivial `false`.

The same is true for:

- the `pgf`'s validity predicate and the call to function `is_valid_parameter_change`
- the `trans_token`'s validity predicate and the call to function `is_valid_parameter`

Problem Scenarios

This finding does not present any security violation. However, it obfuscates the code functionality, making future problems more likely.

Recommendation

We suggest refactoring the code to reflect that

- If the governance proposal passed, the parameter change is allowed, regardless of its content.

- If there was no governance proposal that passed, no parameter change is not allowed.

One possible way to do so would be not to call the `is_valid_parameter` function at all, but return an error already in the corresponding hand of the `match` clause.

Miscellaneous Code Findings

Project	Namada Governance and PGF modules
Type	IMPLEMENTATION PRACTICE
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

In this finding, we describe a number of improvements to the code. Those typically do not affect the functionality, but improve the code readability, make code more robust with respect to future changes, or represent a good engineering practice.

1. The comment above the tally type and the tally name `are inconsistent`: the comment talks about two thirds quorum, the name mentions two fifths. The comment needs to be updated (the logic follows `2/5` quorum && `2/3` majority), likely also the name to be consistent with other names (e.g., `TwoFifthsOverTwoThirds`).
2. The comment above the function `get_proposal_author` is misleading, it says:
`/// Get the code associated with a proposal`
3. The functions and variables related to the committing key are spelled inconsistently, dominantly as `committing_key*`, but sometimes as `commiting_key*` (single `t`). This hinders searchability of the code.
Examples: `get_committing_proposals_prefix`, `committing_key`, `no_committing_key`, `commiting_key`
4. There seems to be a lot of shared concerns when it comes to `is_valid_end_epoch`, `is_valid_activation_epoch`, and `is_valid_start_epoch`, even checking of the same things, e.g., [here](#) the `is_valid_end_epoch` checks for the relation between the `start_epoch` and the `current_epoch`, or the minimum voting period being checked in both `is_valid_end_epoch` and `is_valid_start_epoch`. It would be better if these functions were merged.
5. The error message when [checking](#) for the `proposal_id` of the vote is misleading. It says `"Invalid proposal ID. Expected {pre_counter} or lower, got {proposal_id}"`. In reality (and what is checked in the code), it should be `"Invalid proposal ID. Expected lower than {pre_counter}, got {proposal_id}"`
6. The function `is_valid_voting_window` has a parameter `is_validator`. If set to `true`, the function calls `is_valid_validator_voting_period`. However, this parameter is always set to `false`, and the function `is_valid_validator_voting_period` is only called directly. We recommend refactoring.
7. The `code` calculates PGF steward rewards by distributing a portion of the total supply, based on a predefined inflation rate, among stewards according to their reward distribution percentages. Each percentage is

verified to be under 1, ensuring that no steward can claim more than their allocated share, and the total distributed amount remains within the calculated `pgf_steward_inflation`. This design eliminates the need for additional [error handling](#) for fund exhaustion, as the system inherently prevents over-allocation.

8. The [condition](#) can be directly checked in the `if` statement, improving readability and efficiency without impacting functionality.

The optimized code would look like this:

```
if post_content_bytes.len() > max_content_length {  
    let error = Error::new_alloc(format!(  
        "Max content length {max_content_length}, got {}.",  
        post_content_bytes.len()  
    ));  
    tracing::info!("{error}");  
    return Err(error);  
}
```

This eliminates the intermediate variable `is_valid` and directly evaluates the condition, streamlining the logic.

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.





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.

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:

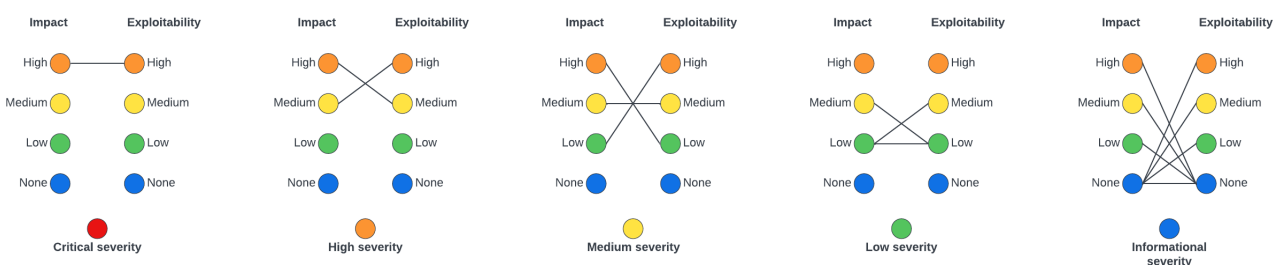
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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

Severity Score	Examples
 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