Squads Security Analysis Report and Formal Verification

certora

October 2023

Table of Contents

Introduction

Solana

Solana is a blockchain platform, in which smart contracts are mainly written in the Rust programming language.
Rust is a safe programming language that implements several novel features centered around providing security guarantees. In that - it nearly fully eliminates several bug-classes, i.e. memory corruption, race-conditions, etc.

Squads

Squads is a DeFi smart contract on the Solana blockchain, implementing a shared-custody wallet, aka a "Multisig", which can contain both fungible and non-fungible tokens.
Deposits may be made into so-called vaults in the Multisig, and payments from these vaults must be approved by a quorum of members through a proposal process. In addition, the Multisig itself is governed by configuration transactions which allow changes to the configuration of the Multisig parameters subject to member consensus.

Audit, Formal Verification

The purpose of this research was to audit the smart contract, with the goal of strengthening any assumptions of soundness, security and robustness of the DeFi application.
In addition, the research was also meant to produce potential candidate properties for formal verification, using Certora's prover.

Timeline

Our audit took place between Aug 29, 2023 and Oct 5, 2023, and included several stages, including an exploratory period, several readings of the code, and hands-on testing.

Methodology

Our experience in embedded and web vulnerability research allowed us to take a slightly different approach than most other auditors in the DeFi ecosystem.

The audit may be divided into two phases:

1. **First reading** - a high level overview of the code base, in which we hunted for "low-hanging fruit" - e.g. arithmetic issues, misconfigured permissions, PDA seed issues and other common pitfalls.

2. **Deep dive and case studies** - a targeted, in-depth analysis of the top attack surfaces, as discovered and characterized in phase 1.

The Squads program

As previously mentioned, the Squads smart contract is a multisig application, in which users may create custodian groups that require approval from multiple members in order to transfer funds, configure the multisig, etc.

The Squads program has several logical units:

1. **Multisig** - a multisig refers to the top level collection of members, and is owned (via PDA) by the Squads program.
2. **Configuration Transaction** - a configuration transaction is a Squads internal transaction type in which the parameters of the multisig are modified.
3. **Vault** - a vault refers to a token storage address, which is owned (via PDA) by the Squads program and a certain multisig. A multisig may have several vaults which are referred to by index.
4. **Vault Transaction** - a vault transaction is a transaction in which funds are paid out from one of the vaults belonging to a multisig.
5. **Batch Transaction** - a batch transaction is a collection of vault transactions which are referenced by a single entity which only needs to be approved once for execution.
6. **Proposal** - a proposal is uniquely associated with a transaction, and may be voted on. More concretely, a transaction may only be executed after an amount of approval votes larger than the consensus threshold has been cast on its associated proposal.
7. **Consensus threshold** - the consensus threshold determines the quorum size required for approving proposals.
8. **Spending limit** - a spending limit is an allowance budget, which permits certain members of the multisig to spend a certain amount of funds in a certain period of time, without requiring a vault transaction and its corresponding approval process.

The code base

We performed our research on the code as presented in the following git repository, commit `81842a7e37143537789e6d17fbe198700f8fe573`:
https://github.com/Squads-Protocol/v4

The Squads code base is written in Rust, and utilizes the Anchor framework, extensively relying on its soundness and security guarantees.

We found the code base to be well-written and concise, properly commentated, organized, and adhering to best-practice standards.

.

Findings

Identifier	Description	Severity	Location
SQ-001	Unintended Signer delegation in *execute_message*	MEDIUM/HIGH	src/utils/executable_transaction_message.rs
SQ-002	Lack of account validation past function boundaries in *realloc_if_needed*	LOW	src/state/multisig.rs
SQ-003	Lack of account validation past function boundaries in *create_account*	LOW	src/utils/system.rs
SQ-004	Incomplete validation of account permissions, resulting in potentially superfluous permissions in *new_validated*, *to_instructions_and_accounts*	LOW	src/utils/executable_transaction_message.rs
SQ-005	Spending limit members are not validated to be part of the multisig in *multisig_add_spending_limit*, *config_transaction_execute*	INFO	src/instructions/multisig_add_spending_limit.rs src/instructions/config_transaction_execute.rs

Main Issues Discovered

SQ-001 - Unintended Signer extension in *execute_message*

In the typical Solana access control model, the privileges afforded to each account participating in the transaction are abstracted to *Signer* and *Writable* privileges.

These privileges are represented in transaction wire format usings 3 small counters:

1. Amount of Signers
2. Amount of Writable Signers
3. Amount of Writable Non Signers

When a transaction is processed by the runtime - these counters along with the preceding signatures are evaluated in order to determine and maintain the privilege status for each account throughout the entire transaction, including into CPI calls.

A nuanced effect of this process is related to the existence of duplicate accounts within a certain instruction.
Under some conditions, when an account is provided twice to the same instruction with two different privilege states, the two states are unified or merged, effectively applying the highest privileges of the two.

The situation where an account is passed twice to the same instruction occurs frequently within the Squads - in vault and batch transaction execution flows.
For clarity, from now on we will refer to the multisig vault/batch execution transaction as the **external transaction**, and to the underlying transaction which was stored during vault transaction creation as the **internal transaction**.

The issue occurs when the `member` account in the external transaction, which pertains to the multisig logic, is also a participant in the underlying transaction logic in the internal transaction, which is a common enough condition.
In this case - since the `member` account is a Signer on the external message, it will implicitly become a Signer for the internal transactions as well.

More concretely, there does not seem to be a way to receive Signer permission for execution of the external transaction without also approving all internal transactions as a side effect.

This can lead to situations where, using minimal social engineering, a member of the multisig may be driven to sign for instructions he did not previously approve, while signing for the execution of a previously approved vault/batch transaction.

We have verified this attack on a localnet according to the following scenario:

1. Create a multisig of threshold 2, with 3 members:
    a. Attacker #1
    b. Attacker #2
    c. Victim
2. Fund each member with 100SOL
3. Fund vault index #0 with 100SOL
4. Create a vault transaction with the following internal instructions:
    a. Transfer 1SOL from vault #0 to Victim
    b. Transfer 99SOL from victim to third-party (non-member)

    The internal instructions do not require Victim to be a Signer.
5. Create a proposal for the vault transaction
6. Attacker #1 and #2 vote to approve, transitioning the proposal to the Approved state
7. Victim executes the approved transaction, causing 99SOL to be removed from his account

If the transaction is executed by any other member of the multisig (e.g. attacker #1/#2), it will fail, which further illustrates the unintended transfer of privilege.

A PoC exploit demonstrating this attack is appended to this document.

In our view, the execution of an approved transaction does not clearly illustrate the potential impact in signing for all internal transactions, and may lead to situations where attackers wrap malicious instructions in ostensibly benign multisig transactions to trick users to signing for them.

We offer the following mitigation options, to be used in some combination:

1. Elucidate the vault/batch execution flow by ensuring members understand the potential impact of signing the execution of a vault

transaction - for instance by adding a UI element in which they must read through and understand the impact of each instruction in the internal transaction.
This will mitigate the social engineering element of the attack.

2. Add additional logic in vault/batch transaction that requires that any instruction added to the internal list be Signer approved at the time of creation - i.e. at the time of creation, when accepting a serialized TransactionMessage - for any instruction that requires signature from a given account, that account must be an additional signer on the on the external transaction.
3. The most comprehensive solution in our view is to reject execution of vault/batch transactions by any member of the multisig which is also a participant in the internal transaction. This solution unfortunately has significantly limiting implications with regards to the multisig's functionality.

SQ-002 - Lack of account validation past function boundaries in *realloc_if_needed*

The function `realloc_if_needed` is responsible for reallocating the multisig account in case of member addition. The flow leading up to it correctly performs account validation - i.e. that the system account is provided. However, the function itself receives AccountInfo structures that do not and cannot automatically perform owner verification.

While no issue arises from this currently, the assumption of validity past function boundaries is a pattern we often find to lead to future mis-use - e.g. a developer adding code to the contract which uses this function assuming it validates the alleged system account's ownership.

As such we suggest adding owner validation inside the `realloc_if_needed` function .

SQ-003 - Lack of account validation past function boundaries in *create_account*`

The create_account function is responsible for creating new accounts in the configuration transaction add spending limit flow.

Similarly to SQ-002, the function assumes its parameters are validated, and they indeed are at the moment.

However as mentioned before we believe this can lead to improper use in future commits and recommend adding an ownership check inside the function as well.

SQ-004 - Incomplete validation of account permissions in *new_validated`to_instructions_and_accounts*

When creating an executable message - which is the data structure pertaining to execution of previously serialized transactions - the code performs validation along the following lines:

1. Each account in the original message must be also passed to the vault/batch execution transaction, in the correct order
2. Each account in the original message classified as Signer, must also be a Signer in the execution transaction
3. Each account in the original message classified as Writable, must also be Writable in the execution transaction

While this is logic is sound insofar as it respects the account permission hierarchy (Signer > Writable Signer > Writable Non-Signer > Read Only), we believe the current validation is insufficient as it permits superfluous permissions to be supplied, relying on the fact that the Solana runtime performs its validation based on the account meta structures.
We recommend that the permissions be explicitly validated to fully match the original instructions - as part of the Squads contract code.
This way the contract is not reliant on relatively opaque behaviors of the Solana runtime with regards to the account meta permission validation.

SQ-005 - Spending limit member validation in *multisig_add_spending_limit*, *config_transaction_executè*

When creating a new spending limit, either via the `MultisigAddSpendingLimit` or `ConfigTransactionCreate/Execute` instructions, a list of members for which the spending limit is applicable is provided.
At the point of creating the spending limit, the addresses in the member list are not validated to be part of the multisig.

They are only validated when “using” the spending limit to transfer funds. We consider this to be an informational only finding, as it does not raise any security issues at the moment but rather deviates from the member validation standards of other similar instructions.

Recommendations

The following do not constitute issues that require immediate fixes in our view, but are still recommended as actions to improve the quality, robustness and soundness of the codebase, and will assist in maintaining its security over time.

SQRC-01 - Explicit "All Destinations" classifier in spending limits

In the current implementation, when creating a spending limit, the creator may specify a list of permitted addresses, to which sending funds is allowed.

If the creator wants to specify that funds may be sent to any destination, he leaves the Destinations vector empty.
While this behavior is correct, we recommend semantically separating the "empty vector" state from the "any destination" state.

For instance:

```
/// ...
    destinations: Option<Vec<Pubkey>>,
/// ...
/// Or alternatively
enum Destination {
    Any,
    Allowed(Vec<Pubkey>),
}
```

While this adds another level of indirection, we believe it may prevent future mis-use of the code at very little computation cost.

SQRC-02 - Disable automatic threshold recalculation when removing members

When removing a member of the multisig as part of a configuration transaction, the code has logic for handling cases where the consensus threshold becomes impossible to pass, thus bricking the multisig.

Since this is a side-effect of the member removal, and not necessarily intended by the executing user - we consider this compound behavior to be non-ideal practice.

Instead, we recommend aborting the transaction in any case where executing it will brick the multisig - including removal of members past the consensus threshold.

The same functionality may be re-created on the client-side (SDK), simply by adding a preceding threshold change action.

SQRC-03 - Invoke invariant validations after each configuration state modification

It is possible to add multiple actions in the same config transaction, but when executing the transaction the invariant validation call is only done at the end.

It means that it is possible to have an intermediate state which is invalid and then reversed, so the invariant check will still pass. For example, add the same use 10 times, and then remove it 10 times, change the threshold to an invalid one and revert it, etc.

While we see no issue currently, we view this as non-ideal practice and recommend calling `invariant()` after each discrete action instead of just at the end of the configuration transaction execution.

SQRC-04 - Add additional rent receiver account in Config Transactions for receiving spending limit rent upon removal

There exist two flows for removal of spending limits - one for a controlled multisig and one for an autonomous multisig.

In the controlled multisig flow, the removal is done using a discrete instruction `MultisigRemoveSpendingLimit` in which the rent collector account is a writable non-signer.

However, in the autonomous multisig flow, the removal is done as part of a configuration transaction, in which the rent collector account is overloaded onto the rent payer account for all the other actions.

As such, it must be a signer, meaning that it holds superfluous permissions in the context of spending limit removal. While this doesn't cause any issues currently, we still recommend separating the rent payer and rent collector accounts in this flow to avoid future confusion.

SQRC-05 - Improve duplicate scanning logic

Currently, the duplication scanning logic depends upon the internal member `members of Multisig` being sorted, in the function `invariant()`.

While the current logic is sound, we believe that such a design is fragile and vulnerable to break with code changes - for instance in case a flow is added or modified such that the list is modified without being sorted after.
We recommend either ensuring that the list is always sorted at the time of duplicate verification, replacing the insertion logic with a sorted-insertion action, or replacing the Vector with a Set-like alternative (e.g HashSet)

SQRC-06 - Improve transaction index based PDA derivation

Transaction PDA derivation relies on the transaction index as one of the seeds. Thus, there exists only one "next" transaction PDA at any time. This leads to the following flow for users:

1. Read the contents of the multisig account and get the current value of the transaction index.
2. Use 'transaction index + 1' as the index when creating their new transaction.

This relies on the network's atomicity model, and may create contention in case two users attempt to create a transaction simultaneously. Only the first will succeed, and the second one will fail since the contract will expect the subsequent transaction index to be used.

As such, this allows a malicious member to attempt so-called "griefing attacks", in which he attempts to create multiple transactions in a row in order to prevent other users from creating new transactions

An option to mitigate this would be to replace the transaction index in the PDA with a random element - e.g. a "create key", similar to how multisigs and spending limits are addressed.
The transaction index should be still stored in the transaction account data itself by the contract, and likewise tallied by the multisig account in order to preserve the same functionality.

Formal verification

The Solana contracts were compiled to SBFv2 using the Rust compiler version 1.67.1. The Solana version was solana-cli 1.14.16.

Notations

✅Indicates the rule is formally verified.
❌Indicates the rule is violated.
⏳Indicates the rule is timing out.

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
- The serialization/deserialization of accounts have been ignored during this verification. We assume that deserialization returns accounts initially filled with arbitrary values and there is no aliasing between any deserialized pointer.
- All CPIs are ignored
- The following functions have been ignored during this verification, and we assume that can return arbitrary values:
  - `find_program_address`
  - `spending_limit_use`
  - `ExecutableTransactionMessage`
    - `new_validated`
    - `execute_message`: verification assumes that `execute_message` cannot call any multisig or proposal related function. This assumption is reasonable due to the use of `protected_accounts`.
- The following functions have been mocked during this verification:
  - `Clock::get()`: returns an arbitrary value but always greater than the last returned value.
  - `vault_transaction_execute`: calls to `ExecutableTransactionMessage::new_validated` and `ExecutableTransactionMessage::execute_message` are ignored by verification.
  - `config_transaction_execute`: verification assumes that account reallocation does not take place and the code of `ConfigAction::AddSpendingLimit` and

`ConfigAction::RemoveSpendingLimit` are ignored by verification.
    - `batch_execute_transaction`: same assumptions as `vault_transaction_execute`.
- Any read to the vector `members` in `Multisig` returns an arbitrary value but the length of the vector is tracked precisely by verification. On the other hand, the vectors `approved`, `rejected`, and `canceled` in `Proposal` are fully modeled precisely (both contents and length) but verification assumes that these vectors cannot be resized. For that, the verification always proves that the length of each of those vectors is less than their capacities so the vectors never need to grow.
- The reallocation of accounts is ignored during verification

Multisig properties (`multisig_create.rs and multisig_config.rs`)

1. ✅ Any function that might modify the multisig (`multisig_create`, `multisig_add_member`, `multisig_remove_member`, `multisig_change_threshold`, `multisig_set_time_lock`, `multisig_set_config_authority`, `vault_transaction_create`, `config_transaction_create`, `batch_create`, and `config_transaction_execute`) calls `invariant` if the function does not revert. (multisig_invariant_function_is_always_called)
2. ✅ After any function that modifies the multisig has been executed, the function `invariant` cannot return any error (multisig_invariants_preserved)
   Extra assumptions: verification ignored `members` permissions and whether `members` has duplicates.
3. ✅ Any function that might modify the multisig consensus (`multisig_add_member`, `multisig_remove_member`, `multisig_change_threshold`, `multisig_set_time_lock`, `multisig_set_config_authority`, and

`config_transaction_execute`) calls `invalidate_prior_transactions` if the function does not revert. (invalidate_prior_transactions_is_always_called)

4. ✅ After any function that modifies the multisig consensus has been executed, `stale_transaction_index` is always equal to `transaction_index` `(`multisig_invalidated_stale_transactions`)`
5. Integrity of the function `multisig_add_member(new_member)` (integrity_of_multisig_add_member)
   - ✅If the function does not revert then the size of `members` increases by one
   - ✅If the function reverts then the size of `members` does not change
6. Integrity of the function `multisig_remove_member(old_member)` (integrity_of_multisig_remove_member)
   - ✅If the function does not revert then the size of `members` decreases by one
   - ✅If the function reverts then the size of `members` does not change
7. ✅ Integrity of controlled multisig: only the config authority can call the functions `multisig_add_member`, `multisig_remove_member`, `multisig_change_threshold`, `multisig_set_time_lock`, and `multisig_set_config_authority`. (integrity_of_controlled_multisig)
8. ✅Integrity of noncontrolled multisig: the multisig config authority must be `Pubkey::default()` (integrity_of_noncontrolled_multisig)

Proposal properties (`batch_add_transaction.rs`, `batch_create.rs`, `batch_execute_transaction.rs`, `config_transaction_create.rs`, `config_transaction_execute.rs`, `proposal_activate.rs`, `proposal_create.rs`, `proposal_vote.rs`, `vault_transaction_create.rs`, `vault transaction_execute.rs`**)**

The following rules:

- invariant_vault_proposal_draft
- invariant_vault_proposal_active
- vault_proposal_active_eventually_active
- vault_proposal_active_eventually_approve
- vault_proposal_active_eventually_reject
- Invariant_vault_proposal_rejected
- invariant_vault_proposal_executed
- invariant_vault_proposal_cancelled
- batch_proposal_draft_eventually_active
- vault_proposal_no_double_approve
- vault_proposal_no_double_reject
- vault_proposal_no_double_cancel

are independent from the type of transaction since they only focus on the proposal voting part. Although we only wrote those rules for one type of transaction they apply to the other transactions.

9. No double voting (vault_proposal_no_double_approve, vault_proposal_no_double_reject, vault_proposal_no_double_cancel)
    - ✅ The same member cannot approve twice the same `Active` proposal
    - ✅ The same member cannot reject twice the same `Active` proposal

.

   - ✅ The same member cannot cancel twice the same `Approved` proposal

10. ✅ If the vault proposal has status `Draft` then it can only be changed to `Active` (invariant_vault_proposal_draft)
11. ✅ If the vault proposal has status `Active` then it can only be changed to `Approved`, `Rejected`, or remains `Active`. (invariant_vault_proposal_active)
   - ✅If the proposal changed to `Approved` then the function `proposal_approve` was the last called function and the size of `approved` is greater or equal than the `threshold` of the multisig.
   - ✅If the proposal changed to `Rejected` then the function `proposal_reject` was the last called function and the size of `rejected` is greater or equal than the cutoff of the multisig.
   - ✅If the proposal changed to `Approved` or `Rejected` then the transaction cannot be stale.
12. ✅ If the vault proposal has status `Active` then it can be eventually changed to `Approved`. (vault_proposal_active_eventually_approved)
13. ✅ If the vault proposal has status `Active` then it can be eventually changed to `Rejected`. (vault_proposal_active_eventually_rejected)
14. ✅ If the vault proposal has status `Active` then it can remain as `Active`, and the size of `approved` is less than the `threshold`, and the size of `rejected` is less than the cutoff of the multisig (vault_proposal_active_eventually_active)
15. ✅ If the vault proposal has status `Approve` then it can only be changed to `Canceled`, `Executed`, or remains `Approve`. (invariant_vault_proposal_approve)
   - ✅If the proposal changed to `Executed` then the function `vault_transaction_execute` was the last called function

- ✅ If the proposal changed to `Executed` then the time that passed between the proposal was `Approved` until it was executed is greater or equal than the `time_lock` of the multisig.
- ✅ If the proposal changed to `Cancelled` then the function proposal_cancel was the last called function and the size of `cancelled` is greater or equal than the multsig `threshold`.
- ✅ The size of `approved` remains greater or equal than the `threshold` of the multisig (i.e., the `approved` vector is not modified even if the proposal is executed or got cancelled)

16. ✅ If the vault proposal has status `Approved` then it can be eventually changed to `Executed` (vault_proposal_approved_eventually_executed)
17. ✅ If the vault proposal has status `Approved` then it can be eventually changed to `Cancelled` (vault_proposal_approved_eventually_cancelled)
18. ✅ If the vault proposal has status `Approved` then it can remain as `Approved` (vault_proposal_approved_eventually_approved)
19. ✅ If the vault proposal has status `Rejected` then the proposal status will not change anymore (invariant_vault_proposal_reject)
20. ✅ If the vault proposal has status `Cancelled` then the proposal status will not change anymore (invariant_vault_proposal_cancelled)
21. ✅ If the vault proposal has status `Executed` then the proposal status will not change anymore (invariant_vault_proposal_executed)
22. ✅ If the batch proposal has status `Approve` then it can only be changed to `Canceled`, `Executed`, or remains `Approve`. (invariant_batch_proposal_approved)
    - ✅If the proposal changed to `Executed` then the function `batch_execute_transaction` was the last called function

- ✅If the proposal changed to `Executed` then the time that passed between the proposal was `Approved` until it was executed is greater or equal than the `time_lock` of the multisig.
- ✅If the proposal changed to `Executed` then the `size` of the batch is equal to `executed_transaction_index`
- ✅If the proposal changed to `Approved` then the `size` of the batch is greater than `executed_transaction_index`
- ✅If the proposal changed to `Cancelled` then the function `proposal_cancel` was the last called function and the size of `cancelled` is greater or equal than the `threshold` of the multisig.
- ✅The size of `approved` remains greater or equal than the `threshold` of the multisig (i.e., the `approved` vector is not modified even if the proposal is executed or got cancelled)

23. ✅ If the batch proposal has status `Approved` then it can be eventually changed to `Executed` (batch_proposal_approved_eventually_executed)
24. ✅ If the batch proposal has status `Approved` then it can be eventually changed to `Cancelled` (batch_proposal_approved_eventually_cancelled)
25. ✅ If the batch proposal has status `Approved` then it can remain as `Approved` (batch_proposal_approved_eventually_approved)
26. ✅ If `batch_add_transaction` does not revert then the size of the batch increases by one, the value of `executed_transaction_index` is 0, and the batch proposal status remains `Draft`. (integrity_of_batch_add_transaction)
27. ✅ If the batch proposal has status `Draft` then it can be eventually changed to Active (batch_proposal_draft_eventually_active)

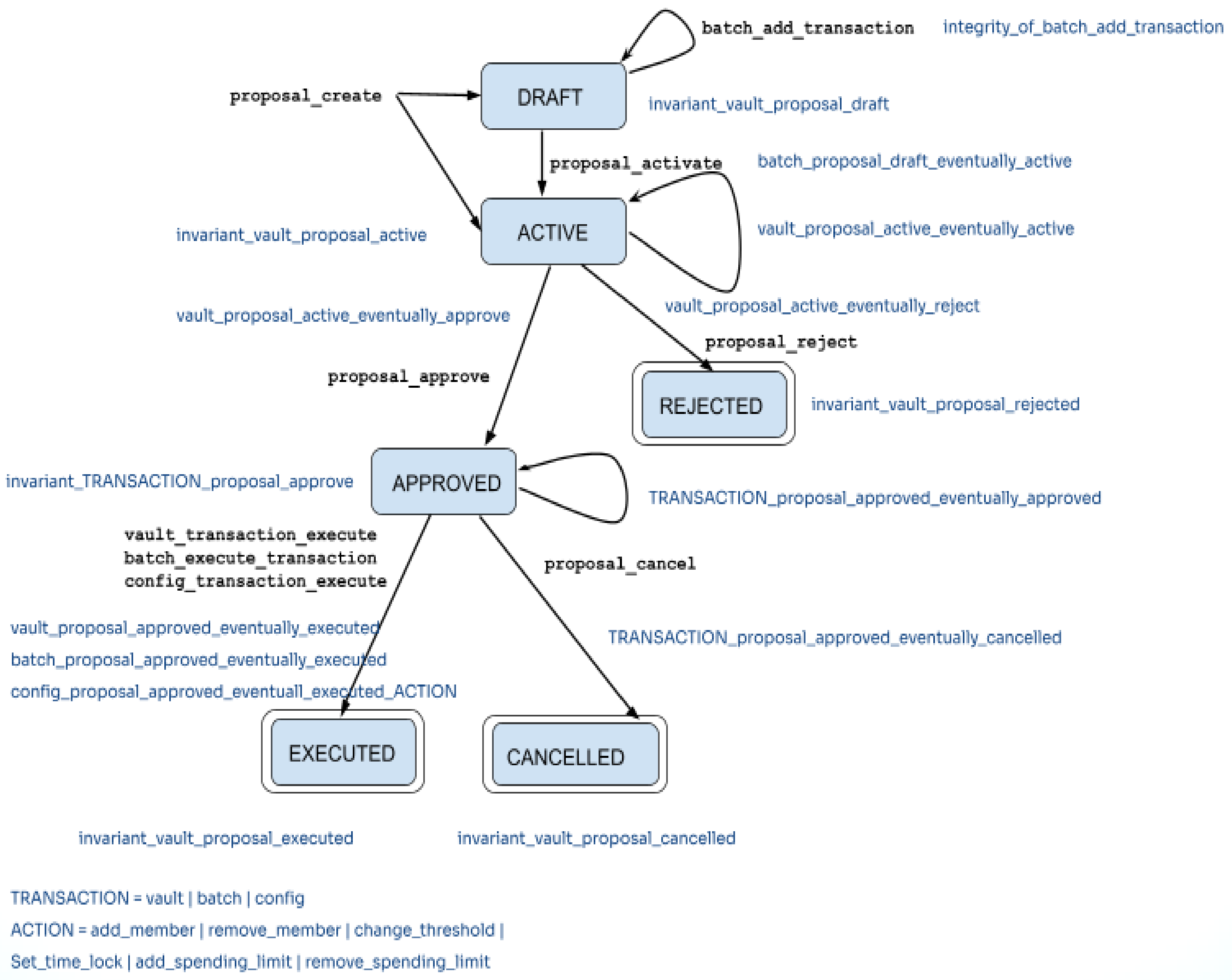
28. ✅ If the config proposal has status `Approve` then it can only be changed to `Canceled`, `Executed`, or remains `Approve` (invariant_config_proposal_approved)
    - ✅If the proposal changed to `Executed` then the function `config_execute_transaction` was the last called function
    - ✅If the proposal changed to `Executed` then the time that passed between the proposal was `Approved` until it was executed is greater or equal than the `time_lock` of the multisig.
    - ✅If the proposal changed to `Executed` then the transaction cannot be stale.
    - ✅If the proposal changed to `Cancelled` then the function `proposal_cancel` was the last called function and the size of `cancelled` is greater or equal than the `threshold` of the multisig.
    - ✅The size of `approved` remains greater or equal than the `threshold` of the multisig (i.e., the `approved` vector is not modified even if the proposal is executed or got cancelled)
29. ✅ If the config proposal has status `Approved` then the proposal status can be eventually changed to `Cancelled`. (config_proposal_approved_eventually_cancelled)
30. ✅ If the config proposal has status `Approved` then the proposal status can remain as `Approved`. (config_proposal_approved_eventually_approved)
31. ✅ If the config proposal has status `Approved` then the proposal status can be eventually changed to `Executed`, and the last executed action is `ConfigAction::AddMember`. (config_proposal_approved_eventually_executed_add_member)
32. ✅ If the config proposal has status `Approved` then the proposal status can be eventually changed to `Executed`, and the last executed

.

action is `ConfigAction::RemoveMember`. (config_proposal_approved_eventually_executed_remove_member)

33. ✅ If the config proposal has status `Approved` then the proposal status can be eventually changed to `Executed`, and the last executed action is `ConfigAction::SetTimeLock`. (config_proposal_approved_eventually_executed_set_time_lock)
34. ✅ If the config proposal has status `Approved` then the proposal status can be eventually changed to `Executed`, and the last executed action is `ConfigAction::ChangeThreshold`. (config_proposal_approved_eventually_executed_change_threshold)
35. ✅ If the config proposal has status `Approved` then the proposal status can be eventually changed to `Executed`, and the last executed action is `ConfigAction::AddSpendingLimit`. (config_proposal_approved_eventually_executed_add_spending_limit)
36. ✅ If the config proposal has status `Approved` then the proposal status can be eventually changed to `Executed`, and the last executed action is `ConfigAction::RemoveSpendingLimit`. (config_proposal_approved_eventually_executed_remove_spending_lim it)

Each state in this automata corresponds to one of the values of `ProposalStatus`. The rules are in blue. We attach each state and transition to one or more rules.


batch_add_transaction
integrity_of_batch_add_transaction
proposal_create
DRAFT
invariant_vault_proposal_draft
proposal_activate
batch_proposal_draft_eventually_active
invariant_vault_proposal_active
ACTIVE
vault_proposal_active_eventually_active
vault_proposal_active_eventually_approve
vault_proposal_active_eventually_reject
proposal_reject
proposal_approve
REJECTED
invariant_vault_proposal_rejected
invariant_TRANSACTION_proposal_approve
APPROVED
TRANSACTION_proposal_approved_eventually_approved
vault_transaction_execute
batch_execute_transaction
config_transaction_execute
proposal_cancel
vault_proposal_approved_eventually_executed
batch_proposal_approved_eventually_executed
config_proposal_approved_eventuall_executed_ACTION
TRANSACTION_proposal_approved_eventually_cancelled
EXECUTED
CANCELLED
invariant_vault_proposal_executed
invariant_vault_proposal_cancelled
TRANSACTION = vault | batch | config
ACTION = add_member | remove_member | change_threshold |
Set_time_lock | add_spending_limit | remove_spending_limit

.