Supra Framework
Smart Contract Assessment Report

Date: June 2, 2025
Version 1.0 [Draft]

Contents

Executive Summary

Supra Labs team engaged with *RektProof Security* to perform an assessment of *Supra Framework*. This assessment was conducted between *Apr 16th* and *May 14th, 2025*.

Assessment overview

This report outlines the results of the security audit conducted on the Supra Framework which defines the standard actions that can be performed on-chain both by the Supra VM.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

Contact Information

Name	Title	Contact Information
Supra Labs		
Arun Doraiswamy	VP of Engineering (Blockchain)	a.doraiswamy@supraoracles.com
RektProof Security		
Naveen Kumar J	Smart Contract Auditor	naina.navs@gmail.com
Surya Prakash Akula	Smart Contract Auditor	d4r3d3v1l.asp@gmail.com
Shiva Shankar	Smart Contract Auditor	oxsh1v@gmail.com

Finding Severity Classification

The following table defines a severity matrix that determines the severity of the bug by evaluating two main factors: the impact of the issue and its likelihood. This approach ensures an accurate assignment of severity.

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Scope

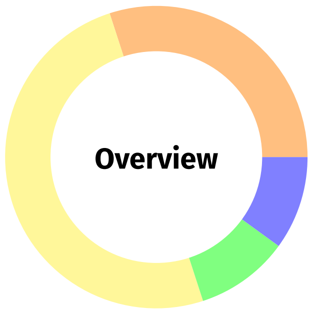
The source code was delivered to us in a git repository at `https://github.com/Entropy-Foundation/aptos-core/blob/aptosvm-v1.16_supra-v1.6.4/aptos-move/framework/supra-framework/`. This audit was performed against commit `24de6e4e76cc67c93a778a77b3169abbda249d03`.

A brief description of the programs is as follows:

Name	Description
Supra Framework	The Supra Framework defines the standard actions that can be performed on-chain both by the Supra VM—through the various prologue/epilogue functions—and by users of the blockchain—through the allowed set of transactions.

Findings

Severity	Count
CRITICAL	0
HIGH	3
MEDIUM	6
LOW	2
INFO	1



0.1 High Risk

0.1.1 Anyone Can Remove Pending Configs and Break Future Updates

Severity: High

Description:

The *config_buffer* module is used to store config changes that should take effect in the next epoch. Trusted modules like *consensus_config*, *execution_config*, *gas_schedule* and *AutomationRegistryConfig*, etc. use this buffer to schedule updates. They add new configs using the upsert() function and later apply them using the extract() function during the *on_new_epoch()* call.

```
/// Upsert an on-chain config to the buffer for the next epoch.
///
/// Typically used in ‘X::set_for_next_epoch()‘ where X is an on-chain config.
public(friend) fun upsert<T: drop + store>(config: T) acquires PendingConfigs {
    let configs = borrow_global_mut<PendingConfigs>(@supra_framework);
    let key = type_info::type_name<T>();
    let value = any::pack(config);
    simple_map::upsert(&mut configs.configs, key, value);
}


/// Take the buffered config ‘T‘ out (buffer cleared). Abort if the buffer is empty.
/// Should only be used at the end of a reconfiguration.
///
/// Typically used in ‘X::on_new_epoch()‘ where X is an on-chaon config.
public fun extract<T: store>(): T acquires PendingConfigs {
    let configs = borrow_global_mut<PendingConfigs>(@supra_framework);
    let key = type_info::type_name<T>();
    let (_, value_packed) = simple_map::remove(&mut configs.configs, &key);
    any::unpack(value_packed)
}
```

Validation steps:

- A new config is added to the buffer using upsert() by a trusted module.
- Before reconfiguration applies it, an attacker calls extract<T>() directly.
- The buffer is now empty, and the config is lost.
- on_new_epoch() checks the config and finds nothing, so the update doesn’t happen. This can be done over and over to block important changes, such as consensus or gas updates.

Impact:

The problem is that *extract()* is marked as *public*, which means that anyone can call it - not just trusted modules. If an attacker calls *extract<T>()* before the real reconfiguration happens, they remove the pending config from the buffer. Then when the trusted module checks if the config exists during *on_new_epoch()*, it won't find anything, and the update will be silently skipped.

Recommendation:

```
public(friend) fun extract<T: store>(): T acquires PendingConfigs {
    let configs = borrow_global_mut<PendingConfigs>(@aptos_framework);
    let key = type_info::type_name<T>();
    let (_, value_packed) = simple_map::remove(&mut configs.configs, &key);
    any::unpack(value_packed)
}
```

Status:

0.1.2 Incorrect Vote Validation in Multisig_Voting Module

Severity: High

Description:

The `multisig_voting` module's `get_proposal_state` function uses an incorrect validation check that only verifies if YES votes meet the threshold, without comparing YES votes against NO votes. While this module is currently only used by `supra_governance` which implements additional safety checks, the module itself is designed to be reusable and could be used by other modules without these safeguards.

```
public fun get_proposal_state<ProposalType: store>(
    voting_forum_address: address,
    proposal_id: u64,
): u64 acquires VotingForum {
    if (is_voting_closed<ProposalType>(voting_forum_address, proposal_id)) {
        let proposal = get_proposal<ProposalType>(voting_forum_address, proposal_id);
        // @audit only checks threshold
        if (proposal.yes_votes >= proposal.min_vote_threshold) {
            PROPOSAL_STATE_SUCCEEDED
        } else {
            PROPOSAL_STATE_FAILED
        }
    } else {
        PROPOSAL_STATE_PENDING
    }
}
```

Validation Steps:

- Create a proposal with 5 voters and threshold of 2
- Vote YES with 2 voters (meeting threshold)
- Vote NO with 3 voters (majority)
- Despite having more NO votes, proposal state returns SUCCEEDED

POC:

```
#[test(supra_framework = @supra_framework, governance = @0x123)]
public entry fun test_multisig_passes_without_majority(
    supra_framework: &signer,
    governance: &signer
) acquires VotingForum {
    account::create_account_for_test(@supra_framework);
    timestamp::set_time_has_started_for_testing(supra_framework);
```

```
let governance_address = signer::address_of(governance);
account::create_account_for_test(governance_address);

let voters = vector[@0xa1, @0xa2, @0xa3, @0xa4, @0xa5];
let proposal_id = create_test_proposal_generic(governance, false, voters);

let proof = TestProposal {};
vote<TestProposal>(
    &account::create_signer_for_test(@0xa1),
    &proof,
    governance_address,
    proposal_id,
    true
);
vote<TestProposal>(
    &account::create_signer_for_test(@0xa2),
    &proof,
    governance_address,
    proposal_id,
    true
);
vote<TestProposal>(
    &account::create_signer_for_test(@0xa3),
    &proof,
    governance_address,
    proposal_id,
    false
);
vote<TestProposal>(
    &account::create_signer_for_test(@0xa4),
    &proof,
    governance_address,
    proposal_id,
    false
);
vote<TestProposal>(
    &account::create_signer_for_test(@0xa5),
    &proof,
    governance_address,
    proposal_id,
    false
);
let TestProposal {} = proof;

timestamp::fast_forward_seconds(VOTING_DURATION_SECS + 1);

assert!(
            get_proposal_state<TestProposal>(
                governance_address,
                proposal_id
            ) == PROPOSAL_STATE_SUCCEEDED,
            1
        );
    }
```

Impact:

The `multisig_voting` module is designed as component that can be integrated by any module requiring voting functionality. While the current implementation in `supra_governance` includes additional safety checks that prevent this vulnerability from being exploited, the core module remains susceptible to misuse. Any future module that directly implements `multisig_voting` without proper vote comparison checks could allow proposals to pass with minority support, as long as they meet the minimum threshold.

Recommendation:

Update the `get_proposal_state` function to match the validation in `voting.move`:

```
if (proposal.yes_votes > proposal.no_votes &&
    proposal.yes_votes >= proposal.min_vote_threshold) {
    PROPOSAL_STATE_SUCCEEDED
} else {
    PROPOSAL_STATE_FAILED
}
```

Status:

0.1.3 Task Registration Fails Due to Incorrect Fee Calculation

Severity: High

Description: When users register automation tasks, the system incorrectly calculates registry occupancy by counting the current task's gas twice, leading to false estimation. The root cause lies in the `register()` function , where `committed_gas` (which already includes the gas of the current task) is passed to the fee estimation function. The estimation function then adds the task gas again, resulting in double counting.

```
    let committed_gas = (automation_registry.gas_committed_for_next_epoch as u128)
                            + (max_gas_amount as u128);
    assert!(committed_gas <= MAX_U64, EGAS_COMMITTEED_VALUE_OVERFLOW);

    let committed_gas = (committed_gas as u64);
    assert!(
        committed_gas <= automation_registry_config.next_epoch_registry_max_gas_cap
    , EGAS_AMOUNT_UPPER);

    // Check the automation fee capacity
    let estimated_automation_fee_for_epoch =
    estimate_automation_fee_with_committed_occupancy_internal(
        max_gas_amount,
        committed_gas,//@audit should pass automation_registry.gas_committed_for_next_epoch
        automation_epoch_info,
        automation_registry_config);
    assert!(
        automation_fee_cap_for_epoch >= estimated_automation_fee_for_epoch
    , EINSUFFICIENT_AUTOMATION_FEE_CAP_FOR_EPOCH);
```

Validation Steps:

- Register a task with 50M gas (50% of 100M registry capacity)
- Verify occupancy should be below 80% congestion threshold
- Observe that congestion fees are incorrectly applied due to false 100% occupancy

POC:

```
#[test(framework = @supra_framework, user = @0x1cafa)]
#[expected_failure(abort_code = EINSUFFICIENT_AUTOMATION_FEE_CAP_FOR_EPOCH, location = Self)]
fun test_incorrect_fee_bug_causes_registration_failure(
    framework: &signer,
    user: &signer
) acquires AutomationRegistry, AutomationEpochInfo, ActiveAutomationRegistryConfig {
    initialize_registry_test(framework, user);
    let task_max_gas = 50_000_000;
let expected_correct_fee =
    AUTOMATION_BASE_FEE_TEST * 50 * EPOCH_INTERVAL_FOR_TEST_IN_SECS / 100;

register(user,
    PAYLOAD,
    EPOCH_INTERVAL_FOR_TEST_IN_SECS + 100,
    task_max_gas,
    20,
    expected_correct_fee,
    PARENT_HASH,
    AUX_DATA
);
}
```

Impact:

The bug makes users charge incorrect fees and prevents legitimate task registrations even when users provide mathematically correct fee amounts.

Recommendation:

Fix the issue by passing the original committed gas without the current task to the fee estimation function:

```
let estimated_automation_fee_for_epoch =
    estimate_automation_fee_with_committed_occupancy_internal(
    max_gas_amount,
    automation_registry.gas_committed_for_next_epoch,
    automation_epoch_info,
    automation_registry_config
);
```

Status:

0.2 Moderate Risk

0.2.1 Missing EVM Configuration Update in Epoch Reconfiguration

Severity: Moderate

Description:

The Supra blockchain uses a reconfiguration process to transition between epochs (time periods). During this process, various system configurations can be updated through a two-step mechanism: first, new configurations are staged in a buffer, then they are applied during the epoch transition.

```
/// Clear incomplete DKG session, if it exists.
/// Apply buffered on-chain configs
///(except for ValidatorSet, which is done inside ‘reconfiguration::reconfigure()‘).
/// Re-enable validator set changes.
/// Run the default reconfiguration to enter the new epoch.
public(friend) fun finish(framework: &signer) {
    system_addresses::assert_supra_framework(framework);
    dkg::try_clear_incomplete_session(framework);
    consensus_config::on_new_epoch(framework);
    execution_config::on_new_epoch(framework);
    supra_config::on_new_epoch(framework);
    gas_schedule::on_new_epoch(framework);
    std::version::on_new_epoch(framework);
    features::on_new_epoch(framework);
    jwk_consensus_config::on_new_epoch(framework);
    jwks::on_new_epoch(framework);
    keyless_account::on_new_epoch(framework);
    randomness_config_seqnum::on_new_epoch(framework);
    randomness_config::on_new_epoch(framework);
    randomness_api_v0_config::on_new_epoch(framework);
    reconfiguration::reconfigure();
}
```

The *evm_config::on_new_epoch()* function is not called in the *reconfiguration_with_dkg::finish()* function, preventing buffered EVM configuration updates from being applied during epoch transitions.

Impact:

- EVM configuration changes initiated through via *evm_config::set_for_next_epoch()* will never take effect
- The EVM will continue operating with outdated configurations indefinitely
- Buffered EVM configurations remain stuck in *config_buffer* without being applied
- Governance decisions regarding EVM parameters cannot be executed

Validation Steps

- Call *evm_config::set_for_next_epoch()* with new configuration bytes.
- Trigger epoch reconfiguration via *supra_governance::reconfigure()*
- Observe that the new EVM configuration is not applied after the epoch transition
- Check that the configuration remains in *config_buffer* instead of being moved to the *EvmConfig* resource.

Recommendation:

Add the missing call to *evm_config::on_new_epoch(framework)* in the *reconfiguration_with_dkg::finish()* function.

Status:

0.2.2 Committee Map Node Transfer Inconsistency Issue

Severity: Moderate

Description:

The *committee_map* module exhibits a critical state inconsistency when nodes are transferred between committees using the *upsert_committee_member* function. Currently, when a node is moved from Committee A to Committee B, the *node_to_committee_map* is updated to reflect the new committee, but the node's entry in Committee A's internal map is not removed. This creates a dangling reference, leading to an inconsistent state where the node appears in both committees' internal storage while officially belonging to only one.

Validation Steps:

```
Before Transfer:
- Committee 100: { map: { NodeA: <data> } }
- Committee 200: { map: {} }
- node_to_committee_map: { NodeA: 100 }

After Faulty Transfer (upsert to Committee 200):
- Committee 100: { map: { NodeA: <data> } }  // Orphaned entry
- Committee 200: { map: { NodeA: <data> } }
- node_to_committee_map: { NodeA: 200 }       // Updated, but old reference remains
```

Impact:

When a node is moved between committees, the system creates an inconsistency where the node's mapping points to the new committee but the node remains in the old committee's member list. This corruption makes the any one of the two committees permanently undeletable because the cleanup process fails when it cannot find the expected node mappings, causing denial of service (DOS).

Recommendation:

Modify *upsert_committee_member* to first remove the node from its old committee (if any) before adding it to the new one.

Status:

0.2.3 Gas Capacity Configuration Desynchronization

Severity: Moderate

Description:

Supra Automation Registry has two gas capacity fields that must remain synchronized: main_config.registry_max_gas_cap and *next_epoch_registry_max_gas_cap*. The code comments explicitly state that these should be identical except during mid-epoch updates.

There is a bug in the *update_config_from_buffer()* function - it updates *main_config.registry_max_gas_cap* but forgets to update *next_epoch_registry_max_gas_cap*. This happens during epoch transitions when these values should be synchronized. Result: the two fields diverge, with different parts of the system using different gas capacity values for the same thing.

Impact:

This bug breaks the fee calculation system and creates economic exploits. Fee calculation functions use *next_epoch_registry_max_gas_cap* (which becomes stale) while capacity validation uses *main_config.registry_max_gas_cap* (which gets updated). Users end up paying fees based on wrong capacity assumptions. Attackers can exploit this by registering tasks during the desync window - they pay fees calculated on old capacity but consume resources based on new capacity.

Recommendation:

Modify the function *update_config_from_buffer()* to properly synchronize both gas-capacity fields during epoch transitions.

Status:

0.2.4 Metadata Params Validation Bypass in FA's mutate_metadata Function

Severity: Moderate

Description:

The fungible asset module implements metadata parameter validations during initial creation through the `add_fungibility` function. These validations enforce important constraints like `MAX_NAME_LENGTH` (32), `MAX_SYMBOL_LENGTH` (10), `MAX_DECIMALS` (32), and `MAX_URI_LENGTH` (512). However, the `mutate_metadata` function, which can modify these parameters post-creation, lacks these validation checks.

```
public fun mutate_metadata(
    metadata_ref: &MutateMetadataRef,
    name: Option<String>,
    symbol: Option<String>,
    decimals: Option<u8>,
    icon_uri: Option<String>,
    project_uri: Option<String>,
) acquires Metadata {
    let metadata_address = object::object_address(&metadata_ref.metadata);
    let mutable_metadata = borrow_global_mut<Metadata>(metadata_address);

    if (option::is_some(&name)){
        mutable_metadata.name = option::extract(&mut name);
    };
    if (option::is_some(&symbol)){
        mutable_metadata.symbol = option::extract(&mut symbol);
    };
    if (option::is_some(&decimals)){
        mutable_metadata.decimals = option::extract(&mut decimals);
    };
    if (option::is_some(&icon_uri)){
        mutable_metadata.icon_uri = option::extract(&mut icon_uri);
    };
    if (option::is_some(&project_uri)){
        mutable_metadata.project_uri = option::extract(&mut project_uri);
    };
}
```

Validation Steps:

- During FA initialization, obtain a `MutateMetadataRef` by calling `generate_mutate_metadata_ref()`
- Initialize the token with valid parameters that pass the initial validation
- Call `mutate_metadata` with values exceeding the limits:
  - name longer than `MAX_NAME_LENGTH` (32)
  - symbol longer than `MAX_SYMBOL_LENGTH` (10)
  - decimals greater than `MAX_DECIMALS` (32)
  - URIs longer than `MAX_URI_LENGTH` (512)
- All these changes will succeed without validation

Impact:

This could cause display issues in wallets and explorers due to excessive lengths in names, symbols, and URIs, while decimal modifications beyond `MAX_DECIMALS` might lead to inconsistent token displays

Recommendation:

Add validation checks in the `mutate_metadata` function that match exactly with those in `add_fungibility`:

Status:

0.2.5 Bypassing the freeze functionality of fungible assets (FA)

Severity: Moderate

Description:

Fungible asset (FA) balances are stored in objects such as `FungibleStore`. If the object storing the balance is deleted, new references to the object cannot be created. However, an exploiter could retain object references, like `ExtendRef`, before deletion and continue to use them even after the object is destroyed. This effectively bypasses administrative freeze capabilities, as the admin with the freeze capability cannot freeze assets if the object storing those assets is deleted. Despite this, the exploiter can still utilize the stored references to withdraw, deposit, and manipulate funds, leaving the admin with no control over the frozen state of the funds.

The root cause of this issue lies in the faulty assumption within the `object::owns()` implementation. It presumed that if an object was marked as "owned" by an address, the object still existed. Exploiters leveraged this by using `ExtendRef` to generate valid signatures, enabling operations such as withdrawals even after the object was deleted.

```
public fun owns<T: key>(object: Object<T>, owner: address): bool acquires ObjectCore {
    let current_address = object_address(&object);

    // @audit Missing existence check before ownership check
    if (current_address == owner) {
        return true
    };

    assert!(
        exists<ObjectCore>(current_address),
        error::not_found(EOBJECT_DOES_NOT_EXIST),
    );
    // ... rest of the function
}
```

Validation Steps:

- Create a fungible store object and obtain references (`Object` and `ExtendRef`)
- Store these references before deleting the object containing `FungibleStore`
- Demonstrate that stored references can still perform operations after deletion

Impact:

The vulnerability allows malicious actors to bypass the freeze functionality of fungible assets by deleting store objects while retaining usable references. This undermines the administrative controls designed to freeze suspicious or compromised accounts, as the stored references can still be used to withdraw and transfer funds even after deletion.

Recommendation:

Update the `object::owns()` implementation to verify object existence before any ownership checks:

```
public fun owns<T: key>(object: Object<T>, owner: address): bool
    acquires ObjectCore {
    let current_address = object_address(&object);

    assert!(
        exists<ObjectCore>(current_address),
        error::not_found(EOBJECT_DOES_NOT_EXIST),
    );

    if (current_address == owner) {
        return true
    };
    // ... rest of the function
}
```

Status:

0.2.6 Committee Type Validation Bypass

Severity: Moderate

Description:

The *Committee map* module defines three committee types with specific node count requirements:

- *FAMILY* committees need more than 1 node
- *CLAN* committees require an odd number of nodes greater than or equal to 3 for Byzantine fault tolerance (2f+1)
- TRIBE committees need node counts in the format 3f+1 with at least 4 nodes

The *validate_committee_type()* function properly enforces these constraints during committee creation through *upsert_committee()*.

However, the vulnerability exists because member removal operations like *remove_committee_member()* do not re-validate committee type constraints after removing nodes. This allows attackers or administrators to create a valid committee that meets type requirements, then systematically remove members until the remaining node count violates the committee type's fundamental consensus requirements.

Validation Steps:

```
// Committee created with correct node count for type
// FAMILY needs >1 nodes
upsert_committee(signer, addr, 1, vector[@0x1, @0x2], ..., FAMILY);

// Later, nodes can be removed one by one without re-validation
remove_committee_member(signer, addr, 1, @0x1);  // Now only 1 node in FAMILY
remove_committee_member(signer, addr, 1, @0x2);  // Now 0 nodes in FAMILY
```

Impact:

This compromises the consensus security and operational integrity of the committee system by allowing committees to exist in states that violate their declared fault tolerance properties.

Recommendation:

Implement committee type re-validation in all member management functions, particularly *remove_committee_member()* and *remove_committee_bulk()*, to ensure that the remaining node count still satisfies the committee type requirements before allowing the removal operation to proceed.

Status:

0.3 Low Risk

0.3.1 Incorrect String Length Validation In *derive_string_concat*

Severity: Low

Description:

The *derive_string_concat* function is intended to enforce a *256-byte* limit on combined prefix and suffix length, as documented in the Move interface.

```
/// aggregator_v2/aggregator_v2.move

/// Arguments passed to concat exceed max limit of 256 bytes (for prefix and suffix together).
const ECONCAT_STRING_LENGTH_TOO_LARGE: u64 = 8;

/// Concatenates ‘before‘, ‘snapshot‘ and ‘after‘ into a single string.
/// snapshot passed needs to have integer type - currently supported types are u64 and u128.
/// Raises EUNSUPPORTED_AGGREGATOR_SNAPSHOT_TYPE if called with another type.
/// If length of prefix and suffix together exceed 256 bytes,
/// ECONCAT_STRING_LENGTH_TOO_LARGE is raised.

/// Parallelism info: This operation enables parallelism.
public native fun derive_string_concat<IntElement>(before: String,
snapshot: &AggregatorSnapshot<IntElement>, after: String): DerivedStringSnapshot;
```

However, the native implementation incorrectly uses *DERIVED_STRING_INPUT_MAX_LENGTH = 1024*, allowing inputs up to 1024 bytes instead of the intended 256 bytes. This means the function accepts inputs that are 4x larger than the design specification, creating a significant gap between intended and actual behavior.

```
/// aggregator_natives/aggregator_v2.rs

/// The maximum length of the input string for derived string snapshot.
/// If we want to increase this, we need to modify BITS_FOR_SIZE in types/src/delayed_fields.rs
pub const DERIVED_STRING_INPUT_MAX_LENGTH: usize = 1024;
```

Impact

When users provide inputs between 256-1024 bytes, these pass validation but can break application logic designed around the 256-byte specification. Systems expecting the intended 256-byte limit may experience buffer overflows or resource exhaustion when processing unexpectedly large 1024-byte inputs.

Recommendation:

```
    pub const DERIVED_STRING_INPUT_MAX_LENGTH: usize = 256;
```

0.3.2 Inefficient Sorting Algorithm Used to Sort Tasks In Automation Registry

Severity: Low

Description:

The *Automation Registry* contract uses bubble sort in *sort_by_task_index()* function, which is not the best algorithm for sorting and is highly inefficient. This inefficient algorithm becomes exponentially more expensive as the registry fills with more tasks. With maximum task capacity, this poor algorithm choice could require a much higher number of operations per epoch transition, potentially over-using the resources and causing transactions to fail completely.

Impact:

This vulnerability can completely paralyze the automation registry system. When the task count grows large enough, epoch transitions will fail due to excessive gas consumption from sorting, preventing the system from processing any tasks, collecting fees, or allowing users to cancel registrations. Attackers can deliberately exploit this by registering many tasks to trigger the DoS condition, effectively locking all user funds in the system since no operations can complete without successful epoch transitions. The system becomes permanently unusable until manual intervention.

Recommendation: Replace the $O(n^2)$ bubble sort with an efficient algorithm

Status:

0.4 Informational

0.4.1 Committee Type Validation Comment Mismatches

Description: In `validate_committee_type` function in `committee_map.move` the comment for CLAN committee type validation states "greater than 3" while the code implements "greater than or equal to 3", creating a documentation-implementation mismatch. The code correctly allows 3-node committees (valid 2f+1 where f=1), but the comment incorrectly suggests these should be rejected. Similarly, for TRIBE committee validation, the comment states "greater than 4" while the code implements "greater than or equal to 4".

```
// Comment incorrectly states: "greater than 3"
// 2f+1, number of nodes in a clan committee should be odd and
greater than 3
assert!(num_of_nodes >= 3 && num_of_nodes % 2 == 1, INVALID_NODE_NUMBERS);

// Comment incorrectly states: "greater than 4"
// 3f+1, number of nodes in a tribe committee should be in the format of 3f+1

and greater than 4
assert!(num_of_nodes >= 4 && (num_of_nodes - 1) % 3 == 0, INVALID_NODE_NUMBERS);
```

Recommendation: Update comments to match implementations:

```
// 2f+1, number of nodes in a clan committee should be odd and greater than or equal to 3
// 3f+1, number of nodes in a tribe committee should be in the format of 3f+1 and greater
than or equal to 4
```

Status: