Security Council multisig and Guardians

TL;DR - Recommendation

Note: this document is a followup from the Kakarot governance document and focuses on the Multisig implementation for the Security Council (SC).

We recommend to implement a custom multisig contract for the SC. It will have simple methods answering our needs to have a multisig with different thresholds depending on the function to call. The Guardians will be part of a Argent multisig.

Introduction

The Kakarot security council is made up of 12 members. They are tasked with safeguarding the security of Kakarot. The Council assesses proposed updates to the Kakarot Core contract and ensure they will not introduce any security risk. In the event of an emergency, the SC has the authority to expedite necessary actions.

In addition to the SC, Guardians are 3 selected members with lesser responsibilities and power: they can pause for a limited time the Kakarot Core contract in case of emergency waiting action from the Security Council.

Composition

The Council will be composed of 12 security and blockchain experts and professionals, all of whom are committed to upholding Kakarot’s values and ensuring the platform's safety. They will collectively be in possession of a multisig key that can take some actions detailed below.

Guardians will be composed of Starkware, Starknet Fondation and Kkrt-labs.

Duties and responsibilities

1. **Security Risk Assessment**: Evaluate proposals to execute Kakarot Core contract function or change of storage, ensuring there are no security risks.
2. **Emergency Response**: Act quickly during emergencies to safeguard the Kakarot protocol.
3. **Transparency**: Provide detailed reports for rejected proposals or emergency interventions.

The security council shall be responsible for assessing the extent to which a security concern is present. Given the nature of such concerns, it is not possible to produce a comprehensive list, so the below list is non-exhaustive and is provided for illustrative purposes only.

- **Smart Contract Vulnerabilities**: Bugs, exploits, and vulnerabilities in smart contracts that can lead to theft of funds, manipulation of the network, or other unintended consequences.
- **Attacks and hacks**: Sybil, Denial of Service (DoS), Double-Spending, Phishing, Social Engineering.
- **Stability and integrity issues**
- **Liveness issues**
- **Oracle manipulation**

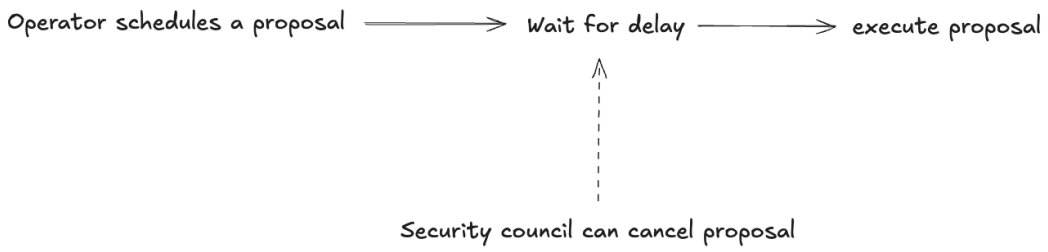
The Security Council's role is limited to ensuring the security of the network, and cannot reject proposal for any reason outside of security.

Governance Flow

Flow	Action	Initiator	Threshold
Regular	Proposal scheduling or execution	Operator	None
Regular	Proposal cancellation	Security Council	50%
Emergency	Emergency proposal execution	Security Council	75%
Emergency	Pausing	Guardians	1 out of 3
Emergency	Pausing	SC	50%
Mitigation	Unpause Malicious Guardian Pausing	Security Council	66%
Mitigation	Revoke a Guardian	Security Council	66%

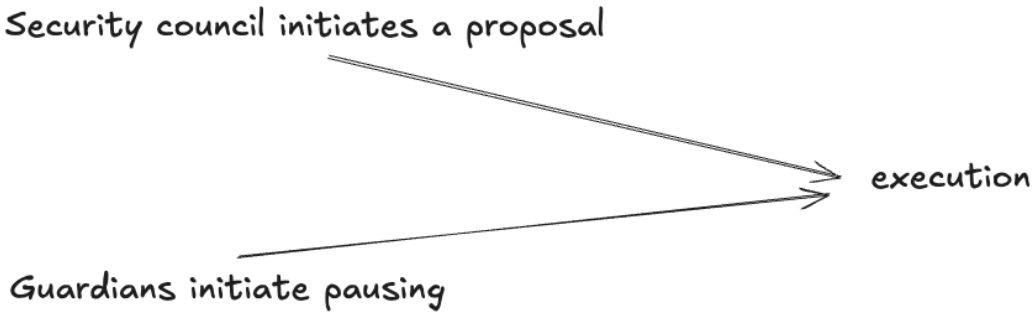
Regular

Each proposals will be submitted through the `ProtocolHandler` (see Kakarot Governance) which managed by a modified `TimelockController` from OpenZeppelin by the `Operator`. The `min_delay` set in the contract needs to pass before being able to execute any proposal. During this period, the Security Council can cancel any proposal it deems not secure.



Emergency

The emergency flow is triggered by the Security Council or by Guardians through the `ProtocolHandler`. In case of time sensitive events, they can trigger proposals without delay.



Veto report from the security Council

In case the Security Council decided to cancel a proposal, it shall be obligated to produce a "Cancellation Report". This report will be made publicly available within fourteen days of cancellation and should provide a detailed account of the following:

1. The rationale behind the negative votes cast by members.
2. An enumeration of the issues identified with the upgrade.
3. Potential avenues for resolving said issues, if feasible.

The Security Council's sole grounds for either rejecting or approving upgrades will be security concerns.

What are the needs of the Kakarot Security Council?

The SC needs to be a Multisig with the possibility to apply different thresholds depending on the action to execute. For Kakarot, it will be translated by having different threshold signatures for specific ProtocolHandler function to execute.

Flow	Action	Threshold
Regular	Proposal cancellation	50%
Emergency	Emergency proposal execution	75%
Emergency	Pausing	50%
Mitigation	Unpause Malicious Guardian Pausing	66%
Mitigation	Revoke a Guardian	66%

Hence the wanted flow for a security council member would be:

1. Opens his Starknet wallet: always the same one used for his SC operations
2. Sign the Tx to execute: needs to target always the same contract
3. Threshold is meet
4. Execution of the Tx

It needs to be the most simple as possible as we do not want an additional mental burden when dealing with critical operations.

What are the current solution ?

Braavos and Argent ❌

Braavos and argent are classical M-out-N multisig. There can be only one threshold configured. Both told us that multiple thresholds by functions could be implemented by plugins but there is no current release date for this.

They also suggested to deploy multiple multisig with different threshold to target the wanted functions.

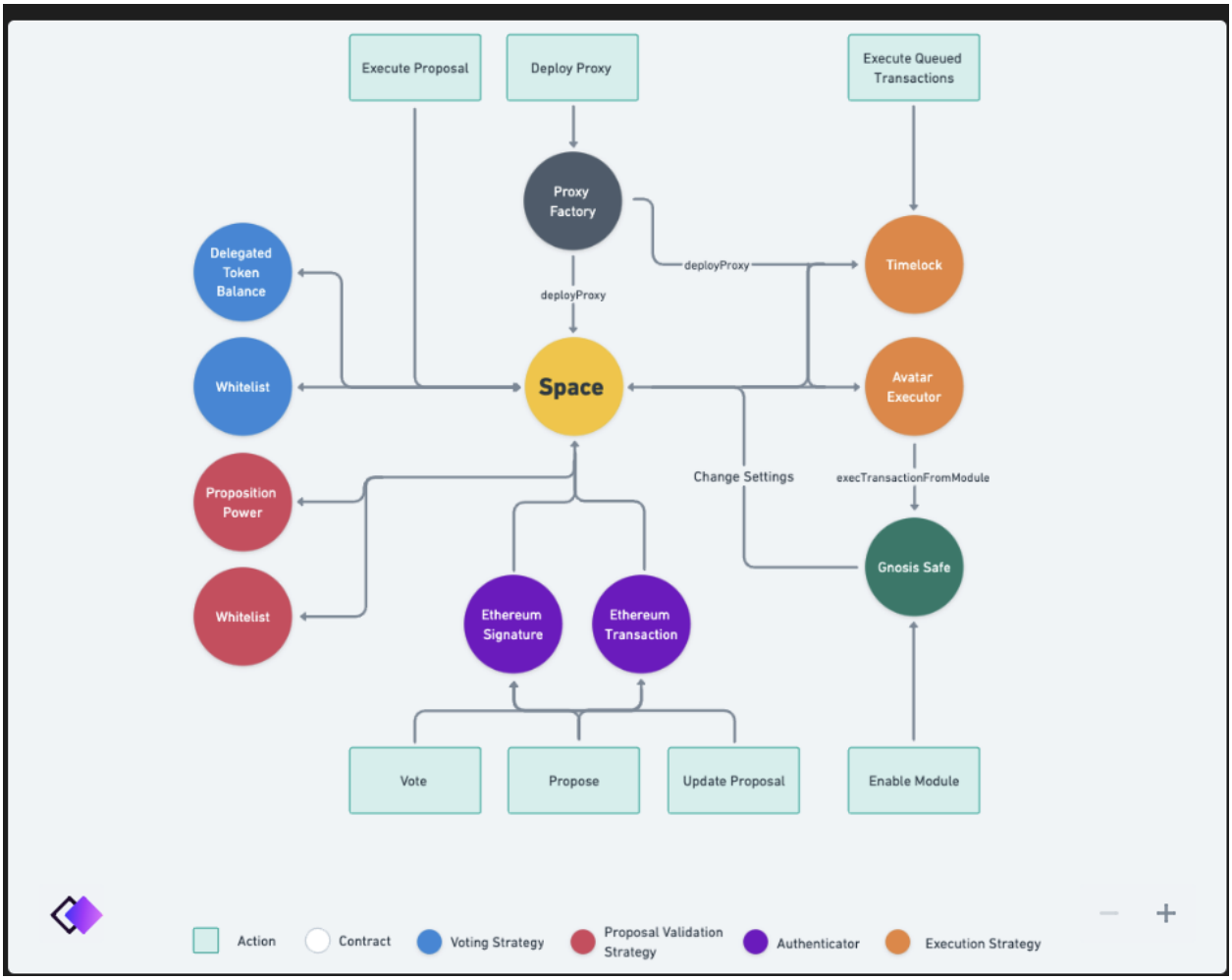
Both suggestion are not acceptable for us: there would be a dependency on the potential release of the feature (not planned due to no demand) and having multiple multisigs for the security council is increasing the ops complexity.

Snapshot box ❌

Snapshot X is an on-chain voting protocol. It possible to create a “space” which is a smart contract to submit proposals. Once the proposal pass, it is forwarded to an “execution strategy”: a contract to execute the proposal according to specific rules.
It would be possible to create a space for the security council to submit proposal to be executed but there are several pain points:
It is not possible to have only one execution strategy with multiple threshold depending on functions to call. We would need to design a custom execution strategy with them and make it audited. Moreover, it will require a custom integration in the UI and the flow would be:

1. submit proposal
2. vote for proposal
3. execute proposal

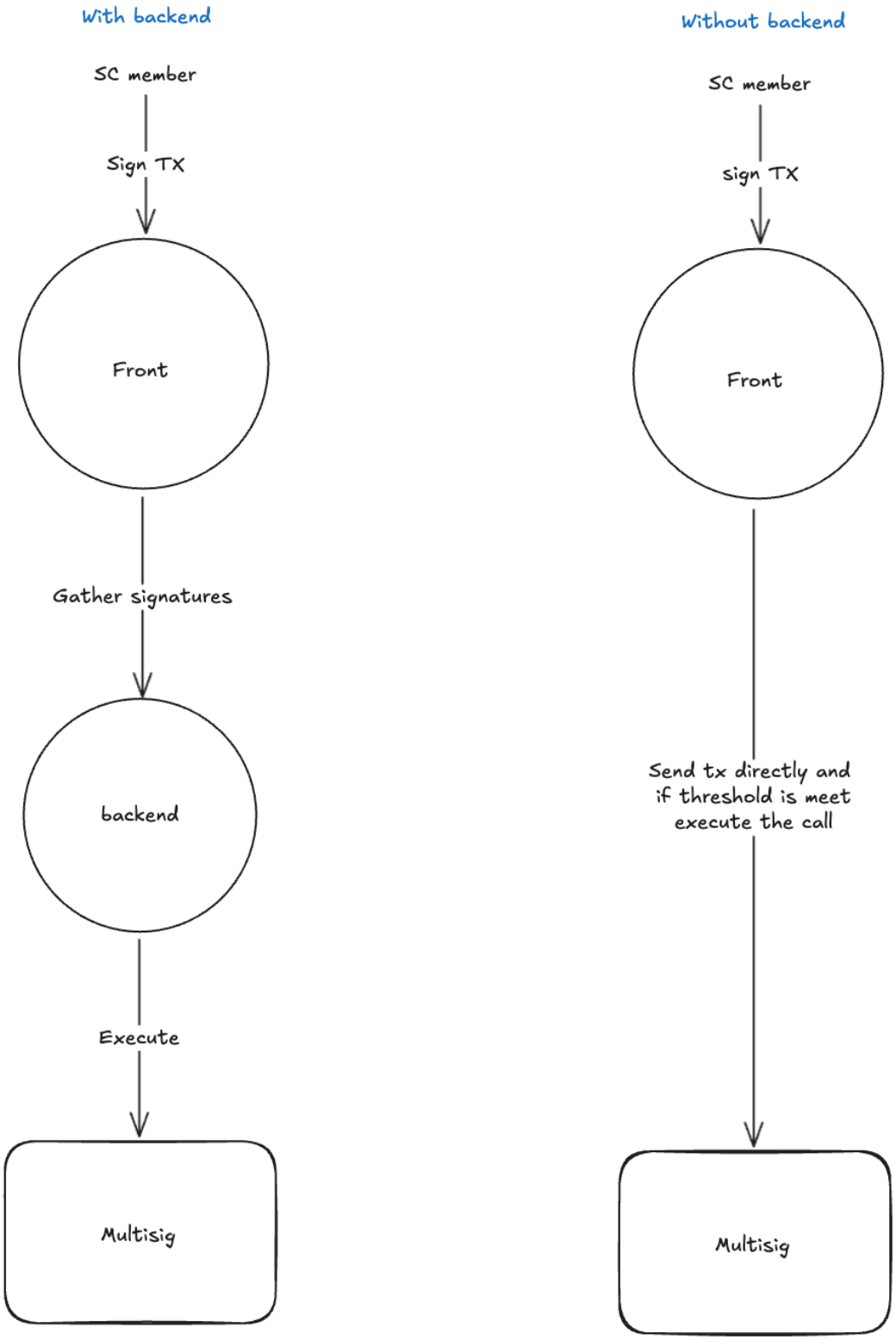
This flow necessitate one more signature than classical multisig as it needs to pass by a proposal submission before execution.



This solution seems not acceptable: the complexity to call a function on the ProtocolHandler is increased. In the end, it is only a 'surcouche' on a custom contract and there will be a dependency on Snapshots for implementation + audits. One advantage is that UI is handled by them and already in place.

Custom ✅

It is possible to design a custom multisig like Zksync to answer our needs. It would be a simple contract that would verify signatures or just receive a tx from each SC member.
It will avoid the drawbacks of Snapshot and answer our needs. Implementation depends on the decision to have a backend or not.



Without backend ✅

It would achieve the same result, will be simpler than snapshot and we are owner of our contracts without dependency.

- each member of the SC need to call the desired function to execute
- It will increase a storage variable corresponding to the tx to execute
- the last submitted tx reaching the threshold will also execute the the call
- No need to maintain a backend

```
// Constants
const pause_threshold = 50%
const emergency_execution_threshold = 75%
const unpause = 66%
const revoke_guardian = 66%
const cancel_proposal_threshold = 50%

// Storage
struct Storage {
  protocol_handler: ContractAddress,
  operator_proposals: ContractAddress,
  members: Vec<ContractAddress>,
  call_hash_by_member: Map<(felt,u32), Map<ContractAddress, felt252>>
};

// External function
fn emergency_execution(valid_until: u64, call: Call);
fn transfer_ownership(valid_until: u64, new_owner: ContractAddress);
fn hard_pause(valid_until: u64);
fn unpause(valid_until: u64);
fn cancel_proposal(valid_until: u64, proposal_id: felt252);

fn change_operator(valid_until: u64, new_operator: ContractAddres);
fn change_security_council(valid_until: u64, new_security_council: ContractAddres);
fn change_gas_price_admin(valid_until: u64, new_gas_admin: ContractAddres);
fn add_guardian(valid_until: u64, guardian_to_add: ContractAddres);
fn remove_guardian(valid_until: u64, guardian_to_remove: ContractAddres);


// in body of function it should look like
// if tx.timestamp > valid_until
//    panic!()
// }
// assert tx.sender in members;
//
// * some other logic *
//
// let hash = poseidon(selector, args)
// assert tx.sender not in call_hash_by_members[hash]
// call_hash_by_members[hash].push(tx.sender)
```

NOTE: this is not a multisig anymore from a conventional standpoint:

- it will require native token to send the transaction
- a member will need to submit the tx instead of just signing it
- several txs will be necessary instead of only one with all the signatures

Could this be a problem in case of emergency / or hack ?
An attacker could monitor the pause function and front run the pause
But guardians will be able to pause with a real multisig.

With backend ❌

It will gather all necessary signatures and send only one tx to execute instead of multiple tx. It needs a backend to be maintained. It is also centralized, if the backend is not up, it will increase the difficulty to send a tx to the multisig. We do not want to do either of those things.

What is our decision ?

The custom solution answer are needs, will be simple without any external dependencies

	Multiple thresholds	Simple/easy to use	audited	ledger HW	Time of dev	Existing UI
Argent	❌	✅	✅	✅	✅	✅
Braavos	❌	✅	✅	✅	✅	✅
Snapshot box	❌ Custom implementation needed	❌	❌ custom implementation	✅	2 weeks custom ui integration + unknown for custom strategy	✅
Custom	✅	✅	needs to be	✅ (Braavos or argent signature to verify)	2 week	❌