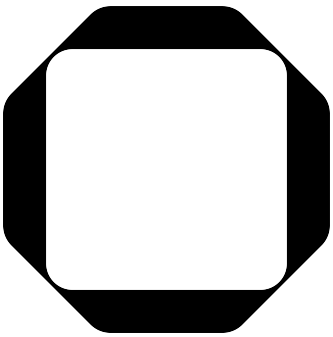


Squads v4

Audit

Presented by:

OtterSec contact@osec.io

Ajay Kunapareddy d1r3wolf@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Squads Protocol engaged OtterSec to perform an assessment of the v4 program. This assessment was conducted between July 6th and September 22nd, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 6 findings in total.

In particular, we addressed noted a number of subtle issues regarding inaccurate CPI account writability semantics (OS-SQD-ADV-00, improper calculation of the space required for reallocating accounts (OS-SQD-ADV-01), and potential denial of service issues when adding transactions to a batch (OS-SQD-ADV-02).

We also made various recommendations concerning validation checks for spending limit members (OS-SQD-SUG-00), range checks for member permissions (OS-SQD-SUG-01), and improved mutable CPI checks to prevent unnecessarily denying transactions (OS-SQD-SUG-02).

Overall, we noted that the code quality of the program was high and the design was solid. The team was also very knowledgeable and responsive to our feedback.

Scope

The source code was delivered to us in a git repository at github.com/Squads-Protocol/v4. This audit was performed against commit 7d79e69. Follow up reviews were performed up to 64af733. The on-chain hash of the audited program is `d48660833989ecea3145ff726164fe640bd90696f03ce00dfd0cda258cbf2fac`.

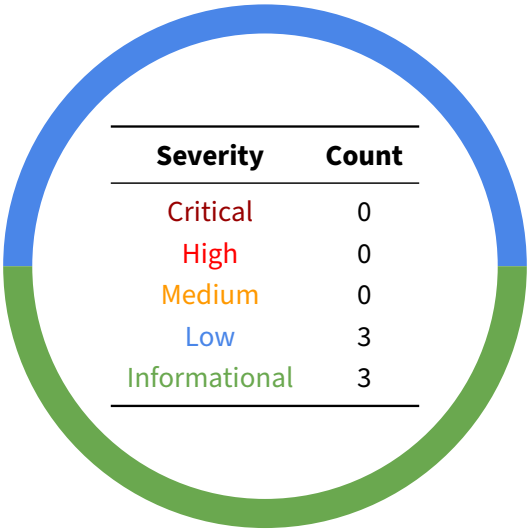
A brief description of the programs is as follows.

Name	Description
v4	As a programmable multisig wallet for Solana, Squads provides the functionality to easily manage developer assets such as programs, tokens, validators, and treasury assets. The multisig enables configurable signature thresholds for the execution of transactions. In addition, a new spending limit feature is introduced, allowing for programmatic limits on token transfers as a first-class feature.

02 | Findings

Overall, we reported 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	0
Low	3
Informational	3

03 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SQD-ADV-00	Low	Resolved	The executor may specify inaccurate writability flags for the underlying transaction in `execute_message`.
OS-SQD-ADV-01	Low	Resolved	While adding members to an account, insufficient reallocation may occur due to incorrect calculations.
OS-SQD-ADV-02	Low	Resolved	Setting the proposal for a batch to an active state may prevent the addition of transactions to the batch.

OS-SQD-ADV-00 [low] | Inaccurate CPI Semantics

Description

When executing a transaction from a Squads multisig, the executor is able to alter the `is_writable` status of accounts.

This is because transactions derive this flag from the passed in `AccountInfo` instead of using the stored data in `VaultTransactionMessage`.

utils/executable_transaction_message.rs RUST

```rust
// `is_signer` cannot just be taken from the account info, because for
↪  `authority`
// it's always false in the passed account infos, but might be true in the
↪  actual instructions.
let is_signer = self.message.is_signer_index(account_index);

let account_meta = if account_info.is_writable {
    AccountMeta::new(*account_info.key, is_signer)
} else {
    AccountMeta::new_readonly(*account_info.key, is_signer)
```

As a result, a malicious multisig executor could violate the intended execution behavior, passing accounts as writable even though the original proposed transaction did not specify writability. Although unlikely, this technically could alter program behavior.

Remediation

Use a writable `AccountMeta` depending on the passed writability specified in `self.message`.

Patch

Resolved in c3d2177 and 2ddacd2 by setting the writable attribute based on `loaded_writable_accounts`.

OS-SQD-ADV-01 [low] | Incorrect Reallocation Size

Description

When adding a member to a multi-signature account, if the pre-allocated memory becomes fully utilized, it is necessary to reallocate. `realloc_if_needed` assesses the requirement for reallocation and performs reallocation for the account, providing additional space for ten members.

The number of members is expanded in two instances:

- `multisig_add_member`.
- `config_transaction_execute`.

While the former only increases the member count by one, the latter may increase the member count by more than one due to the possibility of executing multiple actions. Therefore, if the size of members to add exceeds the sum of the remaining space and the size of ten members, the fixed increment of ten members may result in a shortage of space.

Remediation

Select the larger value between the size increased by ten members and the size increased as needed.

state/multisig.rs DIFF

```
@@ -79,14 +81,17 @@ impl Multisig {
-       let new_size = current_account_size + (10 * Member::INIT_SPACE);
+       let new_size = max(
+           current_account_size + (10 * Member::INIT_SPACE),
+           account_size_to_fit_members,
+       );
```

Patch

Fixed in 5640af0 by taking the larger size for new members or fixed ten members.

OS-SQD-ADV-02 [low] | Blocked Transaction Addition

Description

After creating a batch, adding transactions necessitates a proposal generated by `proposal_create`. The program-derived address of the proposal uses the transaction index utilized in `batch_create` as a seed. When initializing the proposal, the caller may select between the `Draft` and `Active` states.

instructions/proposal_create.rs RUST

```rust
pub proposal: Account<'info, Proposal>,
    pub fn proposal_create(ctx: Context<Self>, args: ProposalCreateArgs) ->
    ↪  Result<()> {
        // ...
        proposal.status = if args.draft {
            ProposalStatus::Draft {
                timestamp: Clock::get()?.unix_timestamp,
            }
        } else {
            ProposalStatus::Active {
                timestamp: Clock::get()?.unix_timestamp,
            }
        };
```

In `BatchAddTransaction::validate`, safeguards permit the addition of transactions to the batch only when the proposal is in a `Draft` state. However, a malicious attacker may set the proposal to `Active`, preventing the utilization of transactions created for the batch.

instructions/batch_add_transaction.rs RUST

```rust
impl BatchAddTransaction<'_> {
    fn validate(&self) -> Result<()> {
        // `proposal`
        require!(
            matches!(proposal.status, ProposalStatus::Draft { .. }),
            MultisigError::InvalidProposalStatus
        );
```

Remediation

Add a cross-program invocation for `proposal_create` at the end of `batch_create` to ensure the creation of a proposal in the `Draft` state for the batch.

Patch

Fixed in 3906ce9 by restricting the proposal creation for multi-signature members with the initiate or vote role.

04 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may lead to security issues in the future.

ID	Description
OS-SQD-SUG-00	Lack of validation checks for members when adding them to `SpendingLimit`.
OS-SQD-SUG-01	When adding members to the multi-signature, verify the range of their permission values.
OS-SQD-SUG-02	Clarify semantics for the incorrect passing of mutable accounts.

OS-SQD-SUG-00 | Lack Of Member Checks

Description

`config_transaction_execute` allows for the configuration of a `SpendingLimit`, regulating the member's transfer regarding the period, vault, amount, destination, and token type. When initializing the `SpendingLimit` account, it would be efficient to ensure that there are no duplicates in the members and that it is not an empty vector.

Remediation

Verify that the length of the vector is greater than zero and check for duplicates within the vector.

instructions/config_transaction_execute.rs DIFF

```
@@ -216,8 +216,12 @@ impl<'info> ConfigTransactionExecute<'info> {
+       let mut members = members.to_vec();
+       // Make sure members are sorted.
+       members.sort();
-       SpendingLimit {
+       let spending_limit = SpendingLimit {
@@ -227,10 +231,14 @@ impl<'info> ConfigTransactionExecute<'info> {
-           members: members.to_vec(),
+           members,
            destinations: destinations.to_vec(),
-       }
-       .try_serialize(&mut &mut spending_limit_info.data.borrow_mut()[..])?;
+       };
+       spending_limit.invariant()?;
+       spending_limit
+           .try_serialize(&mut &mut spending_limit_info.data.borrow_mut()[..])?;
```

state/spending_limit.rs DIFF

```
@@ -62,6 +64,16 @@ impl SpendingLimit {
+   pub fn invariant(&self) -> Result<()> {
+       require!(!self.members.is_empty(), MultisigError::EmptyMembers);
+
+       // There must be no duplicate members, we make sure members are sorted when
    ↪  creating a SpendingLimit.
+       let has_duplicates = self.members.windows(2).any(|win| win[0] == win[1]);
+       require!(!has_duplicates, MultisigError::DuplicateMember);
+
+       Ok(())
+   }
```

Patch

Fixed in 13240af by checking if the vector that includes members is empty or contains duplicates.

OS-SQD-SUG-01 | Missing Permission Range Check

Description

Members of a multi-signature may be one of three types of permissions. These permissions are retained in a bitmask format within the u8 data type. Therefore, as there are three types of permissions, the value of a member's permission should be confined to a range between zero and seven.

state/multisig.rs RUST

```rust
pub enum Permission {
    Initiate = 1 << 0,
    Vote = 1 << 1,
    Execute = 1 << 2,
}
```

Introducing a range check to ensure that the unutilized bit areas are not exceeded would be beneficial when initializing members in `multisig_create` and adding members through `add_member`.

Remediation

Implement a function that validates the permissions and ensure to invoke it whenever adding a member.

state/multisig.rs DIFF

```
@@ -136,6 +136,12 @@ impl Multisig {
+       // Members must not have unknown permissions.
+       require!(
+           members.iter().all(|m| m.permissions.mask < 8), // 8 = Initiate | Vote
    ↪  | Execute
+           MultisigError::UnknownPermission
+       );
```

error.rs DIFF

```
@@ -60,4 +60,6 @@ pub enum MultisigError {
     #[msg("Decimals don't match the mint")]
     DecimalsMismatch,
+    #[msg("Member has unknown permission")]
+    UnknownPermission,
 }
```

Patch

Fixed in 4089d5d by limiting permission to less than eight.

OS-SQD-SUG-02 | Clarify CPI Semantics

Description

CPI semantics regarding the passing of mutable accounts to vault execution are a bit muddled. The underlying issue is that after the `execute_message` concludes, the mutable accounts will deserialize again, meaning any changes during the CPI will be lost.

This is partially checked via the existing `ix.program_id` checks.

utils/executable_transaction_message.rs RUST

```
    if ix.program_id == id() {
        require!(
            ix.data[..8] !=
↪       crate::instruction::VaultTransactionExecute::DISCRIMINATOR,
            MultisigError::ExecuteReentrancy
        );

        require!(
            ix.data[..8] !=
↪       crate::instruction::BatchExecuteTransaction::DISCRIMINATOR,
        );
    }
```

However, these checks are too strict. As an example, a user should be allowed to call execute on an unrelated Multisig.

Remediation

Explicate denylisted accounts and ensure that none of the denylisted accounts are passed as writable.

Patch

Resolved in #24 by explicitly passing in denylisted mutable accounts.

05 | Formal Verification

Overview

Our goal is to formally verify the correctness of the Squads multisig program. We have compiled the code against our formal verification tool. This verification focuses on two main aspects of the program: account and instruction invariants.

The former examines the critical properties of each account, relying on specifying data invariants that are critical to system security. This approach attempts to guarantee that if an entrypoint function starts from a valid state, the resulting state is still valid.

The goal of the latter is to uncover any undesired point of failure within the execution flow that would lead to availability issues and unexpected reverts.

Limitations

It is important to address the brief limitations of this approach. To achieve tractable verification, compromises became necessary. We maintain that these limitations are reasonable and minimally impact the verified logic.

The primary limitation is that our verification scope excludes the serialization or deserialization of Anchor data structures. However, since the program is entirely written in Anchor, an issue here would imply an issue with Anchor's core logic. We do not believe this is a likely risk surface, particularly given the program's relatively standard usage.

In addition, we opted to mock out CPI calls, seed derivation reliant on hashing, and `Clock` time. These aspects represent features within the Solana VM that pose challenges in formal reasoning but can be reasonably abstracted out.

Interesting Invariants

Transactions Non-Malleability

As we began, a crucial invariant we aimed to establish was the non-malleability of multisig transactions.

In simple terms, the goal was to ensure that the proposed transaction executes precisely as outlined in the message, without the possibility for the *executor* to alter any parameters. This is crucial as any ambiguity could allow the executor to potentially deviate from the intended behavior.

Key CPI parameters are specified within the `VaultTransactionMessage` account. However, the executor retains control over the permissions of the provided accounts, which are calculated at runtime during the execution of `execute_message`.

Verifying this invariant required an initial confirmation that the `new_validated` function accurately specifies the accounts into the appropriate vector based on the initial `VaultTransactionMessage`.

To streamline and enhance the accuracy of the verification process, the initial code was modified to include two new vectors: `readonly_accounts` and `writable_accounts`. The former vector includes both static and non-static read-only accounts' Pubkeys, while the latter includes both static and non-static writable accounts' Pubkeys.

Lastly, during the actual invocation, it becomes crucial to demonstrate that the accounts in the `account_infos` possess the correct permissions as specified in the original message. Specifically, a writable account must be classified as either writable or static in a static-writable index, and thus be included in the previously mentioned `writable_accounts` vector. Conversely, a read-only account should be classified as either read-only or static, without belonging to a static-writable index, and should therefore be included in the `readonly_accounts` vector.

utils/executable_transaction_message.rs RUST

```rust
#[cfg(any(kani, feature = "kani"))]
ix.accounts.iter().for_each(|account_meta| {
    kani::assert(
        if account_meta.is_writable {
            self.writable_accounts.contains(&account_meta.pubkey)
        } else {
            self.readonly_accounts.contains(&account_meta.pubkey)
        },
        "Tx non-malleability violated",
    );
});
```

Signature Uniqueness

Another significant invariant we aimed to prove was around the uniqueness of signatures, intending to establish one to one mapping between the multisig and the seeds used in the CPI.

Without this, it could be possible for a multisig transaction to be mistakenly approved by a different multisig. To confirm this link, we need to ensure that each set of seeds includes at least one element that identifies the specific multisig.

Two commonly utilized elements for this purpose are the multisig key, meeting our requirement, and the transaction key derived from the multisig key as its seed, thereby establishing a clear connection. To verify this, we isolated the seed derivation algorithm and wrote the following verification below.

utils/executable_transaction_message.rs RUST

```rust
let vault_seeds = &[
    SEED_PREFIX,
    multisig_key.as_ref(),
    SEED_VAULT,
    &transaction.vault_index.to_le_bytes(),
    &[transaction.vault_bump],
];

let vault_seeds = vault_seeds
    .iter()
    .map(|seed| seed.as_ref())
    .collect::<Vec<&[u8]>>();

let (ephemeral_signer_keys, ephemeral_signer_seeds) =
    derive_ephemeral_signers(transaction_key, &transaction.ephemeral_signer_bumps);

let ephemeral_signer_seeds = &ephemeral_signer_seeds
    .iter()
    .map(|seeds| seeds.iter().map(Vec::as_slice).collect::<Vec<&[u8]>>())
    .collect::<Vec<Vec<&[u8]>>>();

let mut signer_seeds = ephemeral_signer_seeds
    .iter()
    .map(Vec::as_slice)
    .collect::<Vec<&[&[u8]]>>();

signer_seeds.push(&vault_seeds);

signer_seeds.iter().for_each(|seed: &&[&[u8]]| {
    kani::assert(
        seed.contains(&multisig_key.as_ref()) ||
    ↪   seed.contains(&transaction_key.as_ref()),
        "No association",
    );
});
```

Formalized Invariants

Here we provide a complete discussion of all the invariants we verified. This section is split into two parts, account and instruction invariants.

Account Invariants

Multisig

In relation to the `Multisig` account, we proved that the multisig will always be usable. More precisely, it will always have the capability to generate new transactions, due to the presence of the required number of proposers, executors, and voters.

Additionally, we verified the absence of duplicate members and that the threshold always exceeds 0, preventing potential circumvention of the voting process.

We also confirmed that the stale transaction index can never exceed the current transaction index. This ensures that newly created transactions can always be executed, avoiding potential permanent DoS if `stale_transaction_index` were to grow infinitely.

src/state/multisig.rs RUST

```rust
[invariant(
    self.members.len() <= usize::from(u16::MAX)
    && self.threshold > 0
    && !self.members.windows(2).any(|win| win[0].key == win[1].key)
    && self.members.iter().all(|m| m.permissions.mask < 8)
    && Self::num_proposers(&self.members) > 0
    && Self::num_executors(&self.members) > 0
    && Self::num_voters(&self.members) > 0
    && usize::from(self.threshold) <= Self::num_voters(&self.members)
    && self.stale_transaction_index <= self.transaction_index
    && self.time_lock <= MAX_TIME_LOCK
)]
```

SpendingLimit

It was verified that `SpendingLimit` is always valid and meaningful. It must have a minimum of one member and a positive amount. It was also proven that members can't exceed the total spending limit. Additionally, the `last_reset` is always positive, indicating a timestamp, and duplicate members in a `SpendingLimit` are not allowed.

src/state/spending_limit.rs RUST

```
[invariant(
    !self.members.is_empty()
    && !self.members.windows(2).any(|win| win[0] == win[1])
    && self.last_reset >= 0
    && self.remaining_amount <= self.amount
    && self.amount > 0
)]
```

Batch

The `Batch` represents a transaction type where `Multisig` members vote on executing grouped transactions. The `size` variable indicates the batch's total transactions, and `executed_transaction_index` tracks the last executed transaction. It was verified that `executed_transaction_index` cannot exceed the batch size.

src/state/batch.rs RUST

```
[invariant(
    self.size >= self.executed_transaction_index
)]
```

ConfigTransaction

The `ConfigTransaction` manages administrative tasks for the Multisig, including handling member additions and removals, adjusting spending limits, and modifying the vote threshold and timelock.

The account's validity depends on the `Multisig` state during execution and the validity of each action in the actions list, verified during `config_transaction_execute` instruction verification. It was proven impossible to create a `ConfigTransaction` without including actions.

src/state/config_transaction.rs RUST

```
[invariant(
    !self.actions.is_empty()
    && self.actions.iter().all(|action|
        if let ConfigAction::SetTimeLock { new_time_lock, .. } = action {
                *new_time_lock <= MAX_TIME_LOCK
        } else {
            true
        })
)]
```

Proposal

It has been proven that the smart contract's entrypoint prevents reaching an invalid `Approval` state. Specifically, a member cannot vote twice for the same side or vote for opposing sides simultaneously.

src/state/proposal.rs RUST

```
[invariant(
    self.approved.iter().enumerate().all(|(i, &x)|
    ↪  !self.approved[..i].contains(&x))
    && self.rejected.iter().enumerate().all(|(i, &x)|
    ↪  !self.rejected[..i].contains(&x))
    && self.cancelled.iter().enumerate().all(|(i, &x)|
    ↪  !self.cancelled[..i].contains(&x))
    && self.approved.iter().all(|pubkey| !self.rejected.contains(pubkey))
)]
```

Instruction invariants

To test the program's entrypoints, we heavily relied on `succeeds_if` macros. These macros allow us to establish specific constraints regarding the arguments, which would cause failure and lead the call to succeed. If the function still fails, there is a potential underlying issue.

MultisigAddMember

When adding a new member to a `Multisig`, certain conditions must be met for the operation's integrity and security. It is imperative to ensure that the total number of members after insertion does not exceed u16::MAX.

In addition, avoid adding an existing member and validate the permissions of the new member. It is also important to note that the caller must be the `config_authority` to initiate this operation, ensuring only authorized entities can modify the `Multisig` configuration.

src/lib.rs RUST

```rust
[succeeds_if(
    ctx.accounts.multisig.members.len() < usize::from(u16::MAX)
    && ctx.accounts.multisig.members.iter().all(|m| m.key != args.new_member.key)
    && ctx.accounts.system_program.is_some()
    && ctx.accounts.rent_payer.is_some()
    && ctx.accounts.config_authority.key() ==
    ↪  ctx.accounts.multisig.config_authority
    && args.new_member.permissions.mask < 8
)]
```

MultisigRemoveMember

To remove a member from a `Multisig`, they must be a current member. As mentioned above, only the `config_authority` can initiate this operation, ensuring authorized modification of the `Multisig` configuration.

After removal, ensure the `Multisig` remains functional by maintaining at least one member with Execute and Initiate permissions. Additionally, verifying that the number of members with Vote permissions is equal to or greater than the specified threshold for the `multisig` account. This verification guarantees operational continuity and secure functionality post-removal.

src/lib.rs RUST

```
[succeeds_if(
    ctx.accounts.multisig.members.len() > 1
    && ctx.accounts.multisig.members.iter().any(|m| m.key == args.old_member)
    && ctx.accounts.multisig.members.iter().any(|m| m.key != args.old_member &&
    ↪  m.permissions.has(Permission::Execute))
    && ctx.accounts.multisig.members.iter().any(|m| m.key != args.old_member &&
    ↪  m.permissions.has(Permission::Initiate))
    && ctx.accounts.multisig.members.iter().filter(|m| m.key != args.old_member &&
    ↪  m.permissions.has(Permission::Vote)).count()
        >= ctx.accounts.multisig.threshold as usize
    && ctx.accounts.config_authority.key() ==
    ↪  ctx.accounts.multisig.config_authority
    && ctx.accounts.multisig.is_member(args.old_member).is_some()
    && ctx.accounts.multisig.members.windows(3).all(|win| win[0].key != win[1].key
    ↪  && win[0].key != win[2].key)
)]
```

ConfigTransactionCreate

The caller must have Initiate permission on the `Multisig` account. Since this instruction is not supported for controlled multisig accounts, `config_authority` needs to be set to `Pubkey::default()`. Additionally, ensure the multisig's `transaction_index` is less than `u64::MAX` to prevent overflow.

src/lib.rs RUST

```
[succeeds_if(
    !args.actions.is_empty()
    && ctx.accounts.multisig.config_authority == Pubkey::default()
    && ctx.accounts.multisig.member_has_permission(ctx.accounts.creator.key(),
    ↪  Permission::Initiate)
    && ctx.accounts.multisig.transaction_index < u64::MAX
    && args.actions.iter().all(|action|
        if let ConfigAction::SetTimeLock { new_time_lock, .. } = action {
                *new_time_lock <= MAX_TIME_LOCK
        } else {
            true
        })
)]
```

ConfigTransactionExecute

Only a `Multisig` member with Execute permissions can call this instruction on the `Multisig` account. The proposal account must correspond to the designated `Multisig`, have an Approved status, and a minimum duration of the specified `time_lock` duration must have passed since approval.

The transaction must not be stale, indicating that the `transaction_index` must exceed the `stale_transaction_index`. The `Multisig` account's invariants must hold after execution.

We have used a helper function to evaluate the transaction execution. This function checks the transaction's validity and whether the state of the `Multisig` account is consistent with the transaction execution and it will succeed if the transaction is valid.

src/lib.rs RUST

```
[succeeds_if(
    ctx.accounts.multisig.is_member(ctx.accounts.member.key()).is_some()
    && ctx.accounts.multisig.member_has_permission(ctx.accounts.member.key(),
    ↪  Permission::Execute)
    && matches!(ctx.accounts.proposal.status, ProposalStatus::Approved { .. })
    && ctx.accounts.proposal.transaction_index >
    ↪  ctx.accounts.multisig.stale_transaction_index
    && squads_multisig_program::tx_execute_validation_helper(&ctx).is_ok()
)]
```

MultisigChangeThreshold

To update the threshold of a `Multisig` Account, the new threshold must range from 1 to the total number of members with voting permissions. The caller must be the `config_authority` of the corresponding `Multisig` account.

src/lib.rs RUST

```
[succeeds_if(
    ctx.accounts.config_authority.key() == ctx.accounts.multisig.config_authority
    && args.new_threshold > 0
    && args.new_threshold as usize <=
    ↪  ctx.accounts.multisig.members.iter().filter(|m|
    ↪  m.permissions.has(Permission::Vote)).count()
)]
```

SpendingLimitUse

The caller of this instruction must be a member of both the `Multisig` account and the `SpendingLimit` account, with the provided `SpendingLimit` corresponding to the given `Multisig` account. The amount should be below the remaining spending limit, and the spend destination must be a whitelisted address.

There are two cases depending on if this is a native transfer. If the `SpendingLimit` is intended for `lamports` transfers, only the system_program is needed, decimals must equal to 9, and the vault has sufficient lamports.

For SPL token transfers, we must confirm the provided mint matches `SpendingLimit's` mint, `vault_token_account`, `destination_token_account`, and `token_program` must be provided, ensuring the vault has enough tokens.

src/lib.rs RUST

```
[succeeds_if(
    ctx.accounts.multisig.is_member(ctx.accounts.member.key()).is_some()
    && ctx.accounts.spending_limit.members.contains(&ctx.accounts.member.key())
    && ctx.accounts.spending_limit.multisig == ctx.accounts.multisig.key()
    && args.amount <= ctx.accounts.spending_limit.amount
    && (
        ctx.accounts.spending_limit.destinations.is_empty()
        ||
    ↪   ctx.accounts.spending_limit.destinations.contains(&ctx.accounts.destination.key())
    )
    && (
        if ctx.accounts.spending_limit.mint == Pubkey::default() {
            ctx.accounts.mint.is_none()
            && ctx.accounts.system_program.is_some()
            && args.decimals == 9
            && ctx.accounts.vault.lamports() >= args.amount
        } else {
            ctx.accounts.mint.is_some()
            && ctx.accounts.spending_limit.mint ==
    ↪   ctx.accounts.mint.as_ref().unwrap().key()
            && ctx.accounts.vault_token_account.is_some()
            && ctx.accounts.destination_token_account.is_some()
            && ctx.accounts.token_program.is_some()
        }
    )
)]
```

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.