Cyberscope

A TAC Security Company

Audit Report

Mage Labs

June 2025

Repository https://github.com/qu0laz/magelabs-staking

Commit 3cfb6eeb656bfefc7d21b876f48dc3b5100ffaf3

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/qu0laz/magelabs-staking
Commit	3cfb6eeb656bfefc7d21b876f48dc3b5100ffaf3

Audit Updates

Initial Audit	11 Jun 2025

Source Files

Filename	SHA256
./errors.rs	6beebb9cd2c1b0ad6acec2df0fd4da780ebc6b4ca953bd9dc9f9947cbe1530a5
./state/stake_receipt.rs	caf6a2563287ca82642c73bccdcb1031dea583157887400116c3298f68d9e684
./state/mod.rs	27f6ec65b6423d551ddb8c92ef4b2cbc738bbee285be02949c4149c3f36b5e09
./state/stake_pool.rs	2dcebd790a8044cf5f81a2dac0df76b0f241a94a9d9c906d16310d4443caadad
./instructions/claim_reward_tokens.rs	edd3f99d22a393bf9c9ffb296aae899560186426ffc621d77a119a2377b3ebca
./instructions/unstake.rs	7e63798be5b9975ae521c27dac1981b12ae827f39c84c567b3e599b626a90688
./instructions/stake.rs	f1ca1bc956d5a2ab1f52ed26d911a3702ff7df1e2a48647d3adb8ed6a19298af

./instructions/mod.rs	e569b8b8b92046acfc2d8c7ae6ef3ab5ef038ad330b1c5e574976a3dfcd15d6c
./instructions/stake_nft.rs	6fca6bf20e25d3b14df92019c0314c001cd5d9ada317cdc5bf261a364c1b9086
./instructions/withdraw.rs	12995d7186905a8a48e54f70cbbe2dd7f83287f199c7e9c2fd087e3b43b6a214
./instructions/admin/update_authority.rs	bdf60b96dd939e4921a61ef8593b8dfa6d491c3d4c5484c02c1fef294935c213
./instructions/admin/add_token.rs	648f615982999d9634f3e448d3544dbb35bc04f8a2dbf8d2cb59cf3ee7a338d8
./instructions/admin/mod.rs	fe84eff17640e1fb35cb7a753f8818fb2d4648e013e119bff48c8a0beb8b2449
./instructions/admin/create_stake_pool.rs	3eba01b2e13396c7a2db1f0dee69ee768188f938cd4c1a7d2c1306709faa953c
./instructions/admin/add_nft.rs	aaf54106309a0c9ccb370ee71c787cc4547fdde9c4764d9dafe14061430ef741
./instructions/admin/add_reward_pool.rs	d1d49c4e451026741910429e33d4ee07db0a809c94c914e90aefc3f0b5ecf5f4
./instructions/increase_stake.rs	df23d77a365f97e3e0222a110800f0bae8f08d5e386c7f278f95a66dea93413b
./instructions/mint_burn_redeem.rs	7e9b09de769c1414a68e641b1ca976d40a22daf39ff70c0ba5f72e4063c43598
./lib.rs	d9fb80e41046861f2ebc6a5bebb7a5097bf1f268fc5d23c58fae15495805cfc3

./uint.rs	1d842809e43e1ee702390e311492eaef864c8ced7711e81fe31e76c239b70a25
./macros.rs	c7e386afda5354bfa4fa90a15fa4e2841bc216c09294810ded3324fd2015a1ab

Overview

The Mage contracts implement a modular and extensible staking system that supports both fungible tokens and NFTs, allowing users to stake assets in exchange for proportional reward distributions. At its core, the system revolves around the `StakePool` account, which maintains authority, tracks custom asset weights, and manages multiple `RewardPools`. Administrators can initialize stake pools, add supported tokens or NFT collections with specific weights, assign reward mints, and update pool authorities. The reward mechanism ensures that rewards are distributed fairly based on weighted stake contributions, with accounting tokens optionally redeemable for actual reward tokens. The design promotes flexibility, precise reward allocation, and composability with various asset types while enforcing access control and account validation throughout the lifecycle of staking and reward operations.

Admin Functionality

The admin functionality of the protocol enables privileged users to configure and manage the `StakePool` through a set of permissioned instructions. Using `CreateStakePool`, an admin initializes a new pool instance with an assigned authority. The `AddToken` and `AddNft` instructions allow the admin to register new stakeable assets—either fungible tokens with associated vaults or NFT collections verified through Metaplex metadata—each with custom weight parameters influencing stake distribution. Through `AddRewardPool`, the admin defines reward configurations by linking real and synthetic reward mints with vaults and setting mint authorities. Finally, `UpdateAuthority` allows for the transfer of administrative control by updating the `StakePool`'s authority key, ensuring flexible and secure protocol governance. All critical operations are gated by signer-based authority checks and account constraints to ensure only authorized entities can modify pool state.

Stake

Users can stake fungible tokens into the protocol by transferring assets from their wallet into a designated vault managed by the `StakePool`. Upon staking, a `StakeReceipt` is generated, recording the user's effective stake based on asset weighting, the original deposit amount, and a snapshot of current reward accumulators. This receipt enables

future reward claims and governs unstaking eligibility. The protocol also recalculates global rewards upon new deposits to ensure accurate distribution.

Increase Stake

The `IncreaseStake` instruction allows users to add more tokens to an existing stake position. Before increasing their stake, users automatically claim and redeem their accumulated rewards. The additional deposit is converted into an updated effective stake, increasing both the user's and the pool's total weighted stake. The process ensures rewards are settled accurately and state remains consistent before stake growth.

Claim Reward Tokens

This function lets users claim synthetic reward tokens that reflect their share of rewards accumulated over time. The protocol recalculates reward rates based on vault balances and user stake before minting the appropriate amount of synthetic tokens. These synthetic tokens represent a user's reward entitlement and can be tracked or redeemed in a later step.

Mint, Burn, and Redeem

This flow enables users to convert synthetic reward tokens into real reward tokens. The contract mints synthetic rewards, burns them from the user's account, and transfers an equivalent amount of real tokens from the reward vault. This two-step process preserves accounting integrity while ensuring users receive actual value from their earned rewards.

Stake NFT

The `StakeNft` instruction allows users to stake NFTs that belong to verified collections. The contract validates the NFT's metadata and ensures it's part of an approved collection. Upon staking, the NFT is transferred to a vault controlled by the `StakePool`, and a `StakeReceipt` is issued to track the user's contribution. The effective stake is computed based on the NFT asset's weight, and rewards begin accruing accordingly. The user's source token account is closed to reclaim rent once the NFT is secured in the vault.

Unstake

Users initiate the unstaking process using the `Unstake` instruction, which applies to both fungible token and NFT stakes. This operation ensures rewards are up to date by recalculating the pool's reward distribution and minting any outstanding rewards. It then decreases the total weighted stake and updates the user's `withdrawable_at` timestamp, enforcing a cooldown period before the actual withdrawal is allowed. This preserves fair reward distribution and prevents immediate stake-exit abuse.

Withdraw

Once the cooldown period ends, users can execute the `Withdraw` instruction to retrieve their staked tokens or NFTs. The contract validates the stake receipt and, if the asset is an NFT, verifies its metadata again. The staked asset is transferred from the protocol vault back to the user's wallet. If the withdrawn asset is an NFT, the associated vault is closed to clean up and reclaim rent. This instruction finalizes the full lifecycle of a stake and ensures secure asset return to the rightful owner.

Here is a clear and concise paragraph describing how rewards are applied in this system:

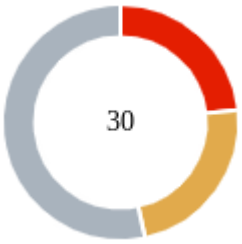
Reward Distribution Mechanism

The reward system distributes tokens to stakers proportionally based on their *effective stake*, which accounts for the weight of the staked asset. When tokens are deposited into a reward vault, the `recalculate_rewards_per_effective_stake` function updates each `RewardPool` 's `rewards_per_effective_stake` accumulator by computing the difference between the current and previous vault balances. This value is scaled and divided by the `total_weighted_stake` to ensure fair allocation. During withdrawal or unstaking, the user's share of rewards is calculated by multiplying the difference in reward-per-stake with their effective stake, then minting accounting reward tokens. If `burn_and_redeem` is enabled, those tokens are burned and equivalent actual rewards are transferred from the vault. This mechanism ensures precision, fairness, and compatibility with both fungible and NFT-based staking assets.

Contract Readability Comment

The audit aimed to assess the contracts for security, correctness, and overall code quality. While the project introduces a functional staking and reward mechanism, the current implementation lacks the robustness and clarity expected from production-grade Solana programs. The codebase shows signs of incomplete logic, insufficient validation, and structural weaknesses that impact both security and maintainability. Overall, the contracts require significant refactoring to meet Solana and Rust best practices. In their current state, the contracts are not considered production-ready, and we recommend a thorough review and rework before deployment.

Findings Breakdown



Critical	7
Medium	7
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	7	0	0	0
Medium	7	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	MRTU	Misaligned Reward Token Usage	Unresolved
Critical	MAC	Missing Access Control	Unresolved
Critical	MRAV	Missing Reward Account Validations	Unresolved
Critical	MRU	Missing Reward Update	Unresolved
Critical	MSTRP	Missing Synthetic Token Redemption Path	Unresolved
Critical	SRRC	Stale Reward Rate Calculation	Unresolved
Critical	UCT	Uninitialized Cooldown Time	Unresolved
Medium	INV	Incomplete NFT Validation	Unresolved
Medium	ISSV	Insufficient Stake State Validation	Unresolved
Medium	MCVL	Missing Custom Validation Logic	Unresolved
Medium	MOTV	Missing Owner Token Validation	Unresolved

●	MSOC	Missing Source Ownership Check	Unresolved
●	MUCE	Missing Unstake Cooldown Enforcement	Unresolved
●	UNWR	Uniform NFT Weighting Risk	Unresolved
●	MRPV	Missing Reward Pool Validation	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	ISU	Inconsistent Signer Usage	Unresolved
●	IRPI	Insecure Reward Pool Input	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MIC	Missing Input Checks	Unresolved
●	MVMV	Missing Vault Mint Verification	Unresolved
●	MZAC	Missing Zero Amount Check	Unresolved
●	NANV	NFT Amount Not Verified	Unresolved
●	POAO	Panic on Arithmetic Overflow	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	RSSC	Redundant Stake State Checks	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved

●	UVP	Unchecked Vault Parameters	Unresolved
●	USU	Unnecessary Struct Usage	Unresolved
●	UAI	Unvalidated Authority Input	Unresolved

MRTU - Misaligned Reward Token Usage

Criticality	Critical
Location	add_reward_pool.rs#49
Status	Unresolved

Description

The contract performs validation and authority logic on the `reward_mint` account, which is expected to represent the actual tokens distributed to users as rewards. However, the implementation inconsistently applies mint authority changes and token issuance logic to the `accounting_reward_mint` instead, which is used solely for synthetic accounting and not for actual token transfers. This creates a critical misunderstanding of the protocol's mechanics. While `reward_mint` appears to be the source of real rewards, the contract neither mints from it nor updates it during reward distribution. Instead, all minting logic is directed at `accounting_reward_mint`, which contradicts its intended purpose and introduces significant confusion. As a result, the code currently validates the supply and freeze authority of `reward_mint`, while setting authority and minting operations on `accounting_reward_mint`, creating an inconsistency that undermines the intended token distribution flow and misleads developers and integrators regarding which mint governs the actual reward logic.

```
impl<'info> AddRewardPool<'info> {
    pub fn validate(ctx: &Context<AddRewardPool>) -> Result<()> {
        // Cannot have freeze authority set
        if ctx.accounts.reward_mint.freeze_authority.is_some() {
            return Err(ErrorCode::FreezeAuthorityMustBeNone.into());
        }

        // Reward mint must have initial supply of 0
        if ctx.accounts.reward_mint.supply > 0 {
            return Err(ErrorCode::InvalidMintSupply.into());
        }
        Ok(())
    }
}

pub fn handler(ctx: Context<AddRewardPool>) -> Result<()> {
    // Update the reward mint's mint authority to the StakePool
    if ctx.accounts.reward_mint.mint_authority !=
COption::Some(ctx.accounts.stake_pool.key()) {
        let cpi_ctx = CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            SetAuthority {
                current_authority: ctx
                    .accounts
                    .accounting_reward_mint_authority
                    .to_account_info(),
                account_or_mint:
ctx.accounts.accounting_reward_mint.to_account_info(),
            },
        );
        set_authority(
            cpi_ctx,
            AuthorityType::MintTokens,
            Some(ctx.accounts.stake_pool.key()),
        )?;
    }
```

Recommendation

It is recommended to align all validation and authority-setting logic with the actual mint used for tracking and distributing rewards. Specifically, if `accounting_reward_mint` is used to mint and manage reward balances, then all checks (such as supply, authority, and freeze restrictions) should be applied to that mint instead of `reward_mint`. Additionally, documentation and variable naming should clearly distinguish between synthetic and real reward mints to avoid protocol misuse or security misunderstandings.

MAC - Missing Access Control

Criticality	Critical
Location	create_stake_pool.rs#30
Status	Unresolved

Description

The contract is missing custom validation logic within the `validate` function of the `CreateStakePool` instruction context. As currently implemented, the function simply returns `Ok(())` without performing any checks, such as verifying that the caller is an authorised admin or ensuring that critical input parameters (e.g., `authority`) are valid. This absence of validation allows **any signer** to initialise a new stake pool, which could lead to unauthorised or malicious pool creation, undermining the intended trust or control model of the protocol.

```
    pub fn validate(_ctx: &Context<CreateStakePool>) ->
Result<()> {
        Ok(())
    }
```

Recommendation

It is recommended to implement explicit access control checks within the `validate` function. For example, ensure that the transaction signer matches a predefined admin address or is otherwise authorised to initialise a stake pool. Additional validation logic should also be added to confirm that key input parameters are valid and not set to default or zero values.

MRAV - Missing Reward Account Validations

Criticality	Critical
Location	stake_pool.rs#331
Status	Unresolved

Description

The `mint_accounting_reward_tokens` function lacks critical runtime checks for several user-supplied accounts, leaving the reward distribution mechanism vulnerable to misdirection or spoofing:

1. Accounting Mint Mismatch: The `accounting_reward_mint_info` account is not validated to match the `reward_pool.accounting_reward_mint` . Without this, attackers could mint synthetic reward tokens to an arbitrary mint.
2. Recipient Account Mismatch: The `owner_accounting_reward_token_info` is not checked for correct ownership (`owner.key()`) or mint consistency (`== accounting_reward_mint_info.key()`), allowing unauthorized recipients or incorrect token types.
3. Burn-and-Redeem Destination Mismatch: In the `burn_and_redeem` branch, the destination SPL token account (`user_reward_token`) is not validated to belong to the caller or match the real `reward_pool.reward_mint` . This opens the door to redirection of real rewards.

```
pub fn mint_accounting_reward_tokens<'info>(
    ...
    let reward_vault_info =
&remaining_accounts[remaining_accounts_index];
    let accounting_reward_mint_info =
&remaining_accounts[remaining_accounts_index + 1];
    let owner_accounting_reward_token_info =
        &remaining_accounts[remaining_accounts_index + 2];
    ...
    token::mint_to(cpi_ctx, total_claimable)?;
    if burn_and_redeem {
    ...
        let user_reward_token =
&remaining_accounts[remaining_accounts_index + 3];
        ...
    }
    remaining_accounts_index += page_size;
...
}
```

Recommendation

It is recommended to add the following validations:

- `require!(accounting_reward_mint_info.key() == reward_pool.accounting_reward_mint, ...)`
- `require!(owner_accounting_reward_token_info.mint == accounting_reward_mint_info.key(), ...)`
- `require!(owner_accounting_reward_token_info.owner == owner.key(), ...)`
- `require!(user_reward_token.owner == owner.key(), ...)`
- `require!(user_reward_token.mint == reward_pool.reward_mint, ...)`

These checks are necessary to enforce correct and secure reward delivery, prevent misdirection of tokens, and preserve the integrity of both synthetic and real reward flows.

MRU - Missing Reward Update

Criticality	Critical
Location	stake.rs#102 unstake.rs#60 stake_pool.rs#292
Status	Unresolved

Description

The contract is designed to distribute rewards proportionally based on the difference between the current `rewards_per_effective_stake` and the user's stored `claimed_amounts` recorded at the time of staking. However, this mechanism critically depends on the admin manually transferring reward tokens into the `reward_vault`. If no transfer occurs between the time a user stakes and later attempts to redeem, the `rewards_per_effective_stake` remains unchanged. As a result, the difference between the updated value and the stored `claimed_amounts` will be zero, and the user will receive no rewards—even if they have been staked for a long period.

This behaviour creates a misleading incentive structure where users expect proportional rewards over time but are entirely dependent on external admin intervention to trigger the accrual logic. The absence of automatic reward accrual or real-time recalculation at stake/unstake time makes the reward system unreliable and potentially unfair to participants.

```
   stake_receipt.owner = ctx.accounts.owner.key();
    stake_receipt.deposit_timestamp = clock.unix_timestamp;
    stake_receipt.claimed_amounts =
stake_pool.get_claimed_amounts_of_reward_pools();
    stake_receipt.effective_stake = effective_stake;
    stake_receipt.mint = ctx.accounts.source.mint;
    stake_receipt.native_amount = args.amount;
    stake_receipt.stake_pool = stake_pool.key();
    stake_receipt.vault = ctx.accounts.vault.key();
    stake_receipt.withdrawable_at = 0;

...
    ctx.accounts.stake_pool.mint_accounting_reward_tokens(
        ctx.accounts.owner.to_account_info(),
        ctx.accounts.stake_pool.to_account_info(),
        ctx.accounts.token_program.to_account_info(),
        &ctx.accounts.stake_receipt,
        &ctx.remaining_accounts,
        Unstake::REMAINING_ACCOUNT_PAGE_SIZE,
        true,
    )?;
```

```
    pub fn mint_accounting_reward_tokens<'info>(
        &self,
        owner: AccountInfo<'info>,
        stake_pool_account: AccountInfo<'info>,
        token_program_info: AccountInfo<'info>,
        stake_receipt: &StakeReceipt,
        remaining_accounts: &[AccountInfo<'info>],
        page_size: usize,
        burn_and_redeem: bool,
    ) -> Result<()> {
        let mut remaining_accounts_index: usize = 0;
        for (index, reward_pool) in self.reward_pools.iter().enumerate()
{
            if reward_pool.is_empty() {
                continue;
            }
            // Calculate the amount of reward tokens the user should get.
            let claimable_per_effective_stake = reward_pool
                .rewards_per_effective_stake
                .as_u128()

.checked_sub(stake_receipt.claimed_amounts[index].as_u128())
                .unwrap();
            // Note: Cannot overflow, 2^128 * 2^128 < 2^256
            let total_claimable =
U256::from(claimable_per_effective_stake)

.checked_mul(U256::from(stake_receipt.effective_stake.as_u128()))
                .unwrap()
                .checked_div(U256::from(SCALE_FACTOR_BASE))
                .unwrap()
                .as_u64();
            msg!(
                "CLAIMABLE {:?} | {:?} | {}",
                claimable_per_effective_stake,
                stake_receipt.effective_stake,
                total_claimable
            );
            if total_claimable == 0 {
                remaining_accounts_index += 1;
                continue;
            }
```

Recommendation

It is recommended to ensure that reward rates (`rewards_per_effective_stake`) are updated dynamically and proportionally to the user's effective stake each time a user stakes or unstakes. This approach, as outlined in the `SRRC - Stale Reward Rate Calculation` finding, would decouple reward distribution from admin funding and make the staking experience more transparent, accurate, and production-safe.

MSTRP - Missing Synthetic Token Redemption Path

Criticality	Critical
Location	claim_reward_tokens.rs#50
Status	Unresolved

Description

The contract mints synthetic accounting tokens (used for tracking rewards) to users during reward claims but lacks a redemption mechanism that allows users to convert or burn these tokens in exchange for the actual underlying reward tokens. As a result, these synthetic tokens accumulate in user accounts without a way to redeem them for value. Additionally, the `mint_burn` logic only handles deltas during reward distribution, not full redemption, further reinforcing the lack of an exit path. This design creates a misleading impression that users have received rewards when, in reality, they hold non-redeemable synthetic balances.

```
// For each reward_pol, mint the reward tokens
    ctx.accounts.stake_pool.mint_accounting_reward_tokens(
        ctx.accounts.owner.to_account_info(),
        ctx.accounts.stake_pool.to_account_info(),
        ctx.accounts.token_program.to_account_info(),
        &ctx.accounts.stake_receipt,
        &ctx.remaining_accounts,
        ClaimRewardTokens::REMAINING_ACCOUNT_PAGE_SIZE,
        false,
    )?;
```

Recommendation

It is recommended to implement a clear and verifiable redemption mechanism that allows users to convert their synthetic reward tokens into real rewards. This should include explicit burn logic tied to minting of real reward tokens, along with proper accounting and validation to prevent abuse. Without such a mechanism, the synthetic rewards model remains incomplete and may confuse users or lead to loss of expected value.

SRRC - Stale Reward Rate Calculation

Criticality	Critical
Location	stake_pool.rs#233
Status	Unresolved

Description

The contract updates the `rewards_per_effective_stake` value only when a change in the token vault's balance is detected (i.e., when tokens are transferred into the reward vault). This logic assumes that rewards are only affected by vault balance updates, but it fails to account for staking-related changes in the pool's `total_weighted_stake`. As a result, if new users stake after reward tokens have been deposited but before `recalculate_rewards_per_effective_stake` is triggered again, they may receive an unfair share of rewards calculated at a stale rate. This undermines reward fairness and introduces opportunities for manipulation.

```
if reward_pool.last_amount == token_account.amount {
    // no change in token account balance, can skip update
    continue;
}
```

Recommendation

It is recommended to trigger `recalculate_rewards_per_effective_stake` on every deposit, withdrawal, and claim, regardless of whether the reward vault balance has changed. Additionally, avoid relying solely on balance deltas to update the reward rate. Incorporating internal accounting events ensures timely and accurate reward distribution aligned with protocol state changes.

UCT - Uninitialized Cooldown Time

Criticality	Critical
Location	unstake.rs#82
Status	Unresolved

Description

The contract uses `unstake_cooldown_time` from the `stake_pool` account to calculate when a user's stake becomes withdrawable. However, there is no enforcement or guarantee that `unstake_cooldown_time` has been initialized to a valid value prior to this calculation. Since this field is left unset and defaulted to zero, the cooldown mechanism becomes ineffective, allowing users to bypass the intended delay before withdrawal. This undermines the staking logic and could enable reward abuse or premature exits from the pool.

```
.unwrap()
.checked_add(ctx.accounts.stake_pool.unstake_cooldown_time)
.ok_or(ErrorCode::ArithmeticError)?;
```

Recommendation

It is recommended to enforce that `unstake_cooldown_time` is explicitly initialized during stake pool setup and validated before use. This can be done by adding runtime checks to ensure it is greater than zero, and by enforcing proper configuration during pool creation or updates. This guarantees that withdrawal timing works as intended and that cooldown logic is not silently bypassed.

INV - Incomplete NFT Validation

Criticality	Medium
Location	add_nft.rs#29
Status	Unresolved

Description

The `validate` function within the `AddNft` instruction performs basic checks for PDA correctness and Metaplex ownership but omits several critical validations specific to NFTs. Notably, it does not verify whether the NFT collection is *verified*, despite such a check being enforced elsewhere in the codebase (e.g., `stake_nft`). Additionally, it does not confirm that the mint being added represents a collection by checking `metadata.collection_details.is_some()`. Lastly, the function fails to validate that the mint has zero decimals—an essential property for distinguishing NFTs from fungible tokens. These gaps can lead to unauthorised or invalid NFT assets being added to the pool, undermining reward logic and pool integrity.

```
impl<'info> AddNft<'info> {
    pub fn validate(ctx: &Context<AddNft>, _args: &AddNftArgs) ->
Result<()> {
        // Validate: Metadata must be owned by Metaplex metadata program
        if ctx.accounts.metadata.owner != &mpl_token_metadata::ID {
            return Err(ErrorCode::InvalidNftMetadata.into());
        }
        // Validate: PDA must match
        let (metadata_pda, _bump) =
Metadata::find_pda(&ctx.accounts.mint.key());
        if ctx.accounts.metadata.key() != metadata_pda {
            return Err(ErrorCode::InvalidNftMetadata.into());
        }

        Ok(())
    }
}
```

Recommendation

It is recommended to enhance the `validate` function to include the following checks:

- Ensure that the NFT's collection is verified (`metadata.collection` exists and `verified == true`).
- Confirm the mint represents a collection (`collection_details.is_some()`).
- Validate that the mint has 0 decimals to guarantee it is a true NFT. These additional validations are necessary to maintain consistency across the contract, enforce proper NFT structure, and mitigate risks from misconfigured or malicious assets.

ISSV - Insufficient Stake State Validation

Criticality	Medium
Location	increase_stake.rs#39 mint_burn_redeem.rs#29 unstake.rs#
Status	Unresolved

Description

Multiple contract instructions rely on internal helper methods (e.g., `can_claim_rewards`, `can_unstake`) to infer that a user has an active or valid stake. However, these checks do not explicitly validate that staking has actually occurred, and the appropriate checks are only handled by the constraints (e.g., `has_one`). Relying solely on structural constraints can lead to false assumptions about a user's eligibility to perform actions like increasing stake or claiming rewards and make the usage of internal functions redundant. This may cause transactions to revert unexpectedly or behave inconsistently across different contract modules.

```
 #[account(
      mut,
      has_one = owner,
      has_one = stake_pool,
      has_one = vault,
    )]
    pub stake_receipt: Account<'info, StakeReceipt>,

    ...
}

impl<'info> IncreaseStake<'info> {
    const REMAINING_ACCOUNT_PAGE_SIZE: usize = 4;

    pub fn validate(ctx: &Context<IncreaseStake>) -> Result<()>
{
        require!(
            ctx.accounts.stake_receipt.can_claim_rewards(),
            ErrorCode::CantClaimRewards
        );
        Ok(())
    }
```

```
    pub fn validate(ctx: &Context<Unstake>) -> Result<()> {
        // Valdiate the StakeReceipt isn't already unstaking
        require!(
            ctx.accounts.stake_receipt.can_unstake(),
            ErrorCode::CantUnstakeAgain
        );
        Ok(())
    }
}
```

Recommendation

It is recommended to implement explicit runtime checks that verify the actual stake state—such as ensuring the staked amount is non-zero, a status flag is set, or the account has been properly initialized through a staking entry point. Avoid depending solely on inferred checks or account relationships, as they may not reliably reflect true staking activity. Consistent and direct validation improves correctness, user experience, and protocol security.

MCVL - Missing Custom Validation Logic

Criticality	Medium
Location	add_token.rs#42
Status	Unresolved

Description

Multiple contracts define a `validate` function for multiple instruction contexts but fail to implement any actual validation logic within them. These functions return `Ok(())` unconditionally, relying solely on attribute-based constraints (such as `has_one = authority`) for enforcing correctness. While these constraints are useful, they do not replace the need for contextual or business-specific validation—such as checking for duplicate resources, verifying configuration bounds, or restricting repeated or unauthorized actions. As a result, the `validate` functions across the codebase are effectively redundant and do not enhance contract security or correctness.

```
#[account(
        mut,
        seeds = [&stake_pool.base.as_ref()],
        bump,
        has_one = authority @ ErrorCode::InvalidAuthority,
    )]
    ...

impl<'info> AddToken<'info> {
    pub fn validate(_ctx: &Context<AddToken>, _args: &AddTokenArgs) ->
Result<()> {
        Ok(())
    }
}
```

Recommendation

It is recommended to review and implement meaningful custom validation logic within each `validate` function to enforce business rules and invariants that cannot be expressed through attribute macros alone. If no additional logic is required, consider removing these empty functions to reduce confusion and maintain code clarity. Consistent and purposeful use of validation functions improves contract robustness and helps prevent subtle misbehaviours.

MOTV - Missing Owner Token Validation

Criticality	Medium
Location	withdraw.rs#16
Status	Unresolved

Description

The contract does not perform runtime validation to ensure that the `owner_token_account` is correctly configured. Specifically, there is no check verifying that the account is owned by the `owner`, nor that it holds the correct token mint associated with the `stake_receipt`. This omission allows users to supply arbitrary token accounts, including accounts they control that use a different mint. As a result, token transfers during the withdrawal process could be redirected to unintended destinations or token types, compromising the integrity and correctness of reward or principal withdrawals.

```
    #[account(mut)]
     pub owner_token_account: Account<'info, TokenAccount>,
```

Recommendation

It is recommended to include the following runtime validations in the `validate` function of the `Withdraw` instruction:

- Ensure that the `owner_token_account.owner` matches the `owner.key()`.
- Ensure that the `owner_token_account.mint` matches the `stake_receipt.mint`.

Adding these validations will ensure that the withdrawal can only be made to a legitimate and expected token account, preserving the integrity of the withdrawal mechanism.

MSOC - Missing Source Ownership Check

Criticality	Medium
Location	stake.rs#63 stake_nft.rs#110
Status	Unresolved

Description

The `transfer_from_payer_to_vault` function in both staking flows (token and nft) performs a transfer from a `source` account to a program-controlled `vault`, using the `payer` as the authority. However, there is no runtime check to ensure that the `source` account is actually owned by the `payer`.

Without this verification, a malicious user could supply a token or NFT account they do not own. Although the transfer will typically fail unless the payer has been granted delegated authority, there are edge cases—such as leftover approvals or explicit delegation—where the transfer may succeed unintentionally. In such cases, if the actual owner of the token account has granted `Approve` access to the payer, the `payer` can unilaterally transfer tokens or NFTs and stake them without consent, effectively stealing or misusing the source's funds.

```
  pub fn transfer_from_payer_to_vault(&self, amount: u64) ->
Result<()> {
        let cpi_ctx = CpiContext::new(
            self.token_program.to_account_info(),
            Transfer {
                from: self.source.to_account_info(),
                to: self.vault.to_account_info(),
                authority: self.payer.to_account_info(),
            },
        );
        token::transfer(cpi_ctx, amount)
    }
```

```
    /// Transfer NFT from the payer's source account to the
vault.
    pub fn transfer_from_payer_to_vault(&self) -> Result<()> {
        let cpi_ctx = CpiContext::new(
            self.token_program.to_account_info(),
            Transfer {
                from: self.source.to_account_info(),
                to: self.vault.to_account_info(),
                authority: self.payer.to_account_info(),
            },
        );
        token::transfer(cpi_ctx, self.source.amount)
    }
```

Recommendation

It is recommended to include an explicit runtime check to ensure that `source.owner == payer.key()` prior to performing the transfer. This validation guarantees that the signer has full control over the funds or NFT being staked and prevents misuse through token delegation or previously approved allowances. Enforcing this constraint makes the staking logic safer, more predictable, and easier to audit.

MUCE - Missing Unstake Cooldown Enforcement

Criticality	Medium
Location	unstake.rs#31
Status	Unresolved

Description

The `Unstake` instruction allows users to unstake without enforcing any minimum cooldown period between staking and unstaking. As a result, a user can stake and immediately unstake, solely to become eligible for reward accrual without maintaining an actual stake. This undermines the staking incentive structure and opens the protocol to potential reward farming abuse—where users repeatedly stake-unstake in rapid succession to extract value without meaningful participation.

```
    pub fn validate(ctx: &Context<Unstake>) -> Result<()> {
        // Valdiate the StakeReceipt isn't already unstaking
        require!(
            ctx.accounts.stake_receipt.can_unstake(),
            ErrorCode::CantUnstakeAgain
        );
        Ok(())
    }
}

pub fn handler<'info>(ctx: Context<'_, '_, '_, 'info, Unstake<'info>>) ->
Result<()> {
    let now = Clock::get()?.unix_timestamp;

    {
    ...
```

Recommendation

It is recommended to enforce a minimum cooldown or lock-in period by validating that the current timestamp exceeds the `withdrawable_at` field set during staking. This ensures that users cannot instantly exit and must adhere to the defined unstake delay. Adding this check improves the fairness and security of the staking mechanism and deters opportunistic exploitation.

UNWR - Uniform NFT Weighting Risk

Criticality	Medium
Location	add_nft.rs#45
Status	Unresolved

Description

The contract applies a fixed weight to each NFT asset without accounting for the collection's total supply or the number of NFTs actually staked. Unlike fungible tokens—where weight reflects the staked amount—NFTs are assigned a flat asset weight, which is then applied uniformly across all individual NFTs. This design leads to disproportionate reward distribution, where each NFT receives an equal share of the total NFT asset weight, regardless of how many NFTs exist or are staked. For example, if the NFT asset weight is set to 400/1000 and 10 NFTs are staked, each NFT effectively receives 400/1000. However, if the collection size is 5000, this approach over-allocates rewards relative to their intended share, potentially resulting in inflation or reward abuse.

```
#[derive(AnchorDeserialize, AnchorSerialize)]
pub struct AddNftArgs {
    pub weight_numerator: u64,
    pub weight_denominator: u64,
}

pub fn handler(ctx: Context<AddNft>, args: AddNftArgs) ->
Result<()> {
    let stake_pool = &mut ctx.accounts.stake_pool;

    let asset_weight = Asset::new(
        &ctx.accounts.mint.key(),
        args.weight_numerator,
        args.weight_denominator,
        None,
        Some(ctx.accounts.metadata.key),
    );
    stake_pool.set_next_asset(asset_weight)?;

    Ok(())
}
```

Recommendation

It is recommended to adjust the NFT asset weight calculation by dividing the assigned asset weight by the total supply (or total staked amount) of NFTs. This would ensure each NFT receives a proportional share of the assigned weight, aligning reward distribution with actual stake representation. Alternatively, separate logic should be implemented for NFT-based assets to normalise their contribution based on collection size, preventing disproportionate allocation of pool rewards.

MRPV - Missing Reward Pool Validation

Criticality	Minor / Informative
Location	stake.rs#56
Status	Unresolved

Description

The `validate` function in the `Stake` instruction does not enforce the presence of at least one reward pool before allowing a user to stake assets. While internal checks such as `get_asset_by_mint` ensure the asset exists, they do not verify whether any reward pool is available to distribute rewards. This omission could lead to a misleading user experience where users are allowed to stake tokens without receiving any rewards, or where the staking operation proceeds under invalid economic conditions.

```
   pub fn validate(_ctx: &Context<Stake>, _args: &StakeArgs) ->
Result<()> {
        // Validation for checking if the mint is in the list of Assets
happens in `get_asset_by_mint`.

        Ok(())
    }
```

Recommendation

It is recommended to add a validation check to ensure that at least one active reward pool exists before allowing staking to proceed. This guarantees that staking actions are meaningful and that reward calculations have valid targets. Adding such validation improves the reliability of the protocol and prevents users from unknowingly interacting with an incomplete or improperly configured reward system.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	lib.rs#17
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
  /* Admin related instructions below */

    #[access_control(CreateStakePool::validate(&ctx))]
    pub fn create_stake_pool(
        ctx: Context<CreateStakePool>,
        args: CreateStakePoolArgs,
    ) -> Result<()> {
        instructions::admin::create_stake_pool::handler(ctx, args)
    }

    #[access_control(AddToken::validate(&ctx, &args))]
    pub fn add_token(ctx: Context<AddToken>, args: AddTokenArgs) ->
Result<()> {
        instructions::admin::add_token::handler(ctx, args)
    }

    #[access_control(AddNft::validate(&ctx, &args))]
    pub fn add_nft(ctx: Context<AddNft>, args: AddNftArgs) -> Result<()>
{
        instructions::admin::add_nft::handler(ctx, args)
    }

    #[access_control(AddRewardPool::validate(&ctx))]
    pub fn add_reward_pool(ctx: Context<AddRewardPool>) -> Result<()> {
        instructions::admin::add_reward_pool::handler(ctx)
    }

    #[access_control(UpdateAuthority::validate(&ctx))]
    pub fn update_authority(
        ctx: Context<UpdateAuthority>,
        args: UpdateAuthorityArgs,
    ) -> Result<()> {
        instructions::admin::update_authority::handler(ctx, args)
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

ISU - Inconsistent Signer Usage

Criticality	Minor / Informative
Location	create_stake_pool.rs#11
Status	Unresolved

Description

The contract is using both `payer` and `base` as `Signer` accounts without enforcing that they are the same address. This allows two different signers to be passed in, which can lead to confusion, unintended behaviour, or privilege escalation if `base` is assumed to be the creator or sole controller of the pool.

```
    pub payer: Signer<'info>,

     pub base: Signer<'info>,
```

Recommendation

It is recommended to add a runtime check ensuring that `payer` and `base` are the same signer, or to explicitly document and validate the intended distinction between their roles.

IRPI - Insecure Reward Pool Input

Criticality	Minor / Informative
Location	stake.rs#86 increase_stake.rs#73
Status	Unresolved

Description

The contract relies on `ctx.remaining_accounts` to pass in all relevant `RewardPool` accounts for recalculating reward distribution. This design delegates responsibility to the caller to supply the correct accounts in the correct order, which introduces risks of misconfiguration or intentional manipulation. If the wrong set or sequence of accounts is provided, reward recalculation could behave incorrectly, leading to misallocated rewards, incorrect accounting, or silent failures that are difficult to detect on-chain.

```
    let stake_pool = &mut ctx.accounts.stake_pool;
     stake_pool.recalculate_rewards_per_effective_stake(
         &ctx.remaining_accounts,
         Stake::REMAINING_ACCOUNT_PAGE_SIZE,
     )?;
```

Recommendation

It is recommended to fetch or derive all relevant `RewardPool` accounts internally or through deterministic means instead of relying on user-supplied remaining accounts. If that is not feasible, implement strict validation logic to verify that the supplied accounts match the expected reward pools both in content and order. This ensures that reward calculations operate on trusted data and preserves the integrity of the staking and distribution process.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	create_stake_pool.rs#41 update_authority.rs.rs#29 mint_burn_redeem.rs#38
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
pub fn handler(ctx: Context<CreateStakePool>, args: CreateStakePoolArgs)
-> Result<()> {
...
}
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MIC - Missing Input Checks

Criticality	Minor / Informative
Location	add_token.rs#54 add_nft.rs#51 add_reward_pool.rs#66
Status	Unresolved

Description

The contract fails to validate inputs during the `AddToken` instruction, particularly the `weight_numerator` and `weight_denominator` values used in constructing an `Asset`. There are no checks to ensure the denominator is non-zero, which can lead to division-by-zero panics or silent misbehaviour. Additionally, there is no constraint enforcing `numerator ≤ denominator`, which could result in invalid or misleading weight configurations. The contract also lacks safeguards to prevent the same token (mint) from being added multiple times, which may allow duplication of assets and unexpected vault behaviour.

```
#[derive(AnchorDeserialize, AnchorSerialize)]
pub struct AddTokenArgs {
    pub weight_numerator: u64,
    pub weight_denominator: u64,
}

pub fn handler(ctx: Context<AddToken>, args: AddTokenArgs) ->
Result<()> {
    let stake_pool = &mut ctx.accounts.stake_pool;
    let asset_weight = Asset::new(
        &ctx.accounts.mint.key(),
        args.weight_numerator,
        args.weight_denominator,
        Some(&ctx.accounts.vault.key()),
        None,
    );
    stake_pool.set_next_asset(asset_weight)?;

    Ok(())
}
```

Recommendation

It is recommended to implement explicit validation logic in the `validate` function or within the handler itself to ensure that:

- `weight_denominator` is non-zero,
- `weight_numerator ≤ weight_denominator`, and
- the given asset token has not already been added to the stake pool. These checks are essential for maintaining logical consistency, preventing runtime errors, and enforcing intended economic constraints.

MVMV - Missing Vault Mint Verification

Criticality	Minor / Informative
Location	withdraw.rs#33
Status	Unresolved

Description

The contract does not validate that the `vault` token account contains the correct token mint that matches the `stake_receipt.mint`. While the `vault` address itself is linked to the `stake_receipt` via a `has_one` constraint, the actual contents of the `vault` —specifically the `mint` —are not verified. This creates a risk of misrouted or invalid token transfers, where tokens of an unexpected type are sent to users during withdrawals.

```
    #[account(mut)]
     pub vault: Account<'info, TokenAccount>,
```

Recommendation

It is recommended to add a runtime check that asserts `vault.mint == stake_receipt.mint`. This ensures the vault holds the expected asset and prevents potential misdirection of funds due to mint mismatches. Verifying this strengthens the correctness and reliability of token handling in the protocol.

MZAC - Missing Zero Amount Check

Criticality	Minor / Informative
Location	stake.rs#77 claim_reward_tokens.rs#30
Status	Unresolved

Description

The `StakeArgs` struct allows users to specify the amount of tokens to stake, but the `handler` function does not perform a check to ensure that the provided `amount` is greater than zero. As a result, users can submit staking transactions with a zero value, which may lead to unnecessary on-chain operations, misleading accounting, or unexpected protocol behaviour. In some cases, it may even be exploited to trigger downstream logic (e.g., reward calculations or state updates) without contributing any stake.

```
pub struct StakeArgs {
    /// Amount of tokens to stake
    pub amount: u64,
}

pub fn handler(ctx: Context<Stake>, args: StakeArgs) ->
Result<()> {
    ctx.accounts.transfer_from_payer_to_vault(args.amount)?;
...
```

Recommendation

It is recommended to include a validation step that explicitly checks `amount > 0` before proceeding with staking logic. Rejecting zero-amount transactions prevents wasteful execution, ensures protocol state remains meaningful, and protects against edge-case abuse where zero-stake interactions could influence system state or reward calculations.

NANV - NFT Amount Not Verified

Criticality	Minor / Informative
Location	stake_nft.rs#132
Status	Unresolved

Description

The contract does not validate that the NFT source account holds exactly one token before initiating staking, nor does it confirm that the account balance is zero before attempting to close it. In the context of NFTs, the source account should have an amount of exactly 1 prior to transfer, and 0 afterwards. Failing to enforce these conditions may result in incorrect staking behaviour, runtime errors, or account closure attempts on non-empty accounts, which will cause the transaction to fail unexpectedly. This opens the door to both unintentional bugs and potential abuse of the staking flow.

```
pub fn handler(ctx: Context<StakeNft>) -> Result<()> {
    // Should be 1 for an NFT, but probably better to use source
amount
    let source_amount = ctx.accounts.source.amount;
    ctx.accounts.transfer_from_payer_to_vault()?;

    ...

    ctx.accounts.close_source_account()?;

    Ok(())
}
```

Recommendation

It is recommended to add runtime checks to ensure:

- `ctx.accounts.source.amount == 1` before initiating the transfer (to enforce NFT semantics), and
- `ctx.accounts.source.amount == 0` before calling `close_source_account()` (to ensure the account is eligible for closure).

These checks improve contract safety, uphold NFT logic assumptions, and prevent invalid state transitions or staking errors.

POAO - Panic on Arithmetic Overflow

Criticality	Minor / Informative
Location	stake.rs#113 stake_pool.rs#105
Status	Unresolved

Description

The contract performs arithmetic operations using `.checked_add()` and similar methods but handles potential overflows by calling `.expect()` with a panic message. This approach causes the program to terminate abruptly if an overflow occurs, rather than handling the condition gracefully. Using panics in a smart contract context reduces reliability, obscures failure causes, and increases the risk of denial-of-service scenarios—particularly when user input or edge cases lead to unexpected overflows.

```
    let total_weighted_stake = stake_pool
          .total_weighted_stake
          .as_u128()
          .checked_add(effective_stake.as_u128())
          .expect("overflow");
...
     pub fn calculate_effect_stake(&self, amount: u64) -> u128 {
          let numerator =
primitive::u128::from(self.weight_numerator);
          let denominator =
primitive::u128::from(self.weight_denominator);
          let weight = primitive::u128::from(amount)
               .checked_mul(numerator)
               .expect("overflow")
               .checked_div(denominator)
               .expect("overflow");
          u128(weight.to_le_bytes())
     }
```

Recommendation

It is recommended to replace panic-based overflow handling with proper error propagation using `?` and a descriptive error code (e.g., `ErrorCode::Overflow`). This ensures the contract can fail safely and predictably, improving its robustness and making it easier to diagnose and correct issues during development and runtime.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	stake.rs#63 increase_stake.rs#48
Status	Unresolved

Description

The `transfer_from_payer_to_vault` functions are used to transfer a specified amount of tokens to the contract. The fee or tax is an amount that is charged to the sender of a token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
    pub fn transfer_from_payer_to_vault(&self, amount: u64) ->
Result<()> {
        let cpi_ctx = CpiContext::new(
            self.token_program.to_account_info(),
            Transfer {
                from: self.source.to_account_info(),
                to: self.vault.to_account_info(),
                authority: self.payer.to_account_info(),
            },
        );
        token::transfer(cpi_ctx, amount)
    }
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that a token transfer tax is not a standard feature of the token specification, and it is not universally implemented by all token contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

RSSC - Redundant Stake State Checks

Criticality	Minor / Informative
Location	stake_receipt.rs#36
Status	Unresolved

Description

The `StakeReceipt` struct defines two separate functions— `can_unstake()` and `can_claim_rewards()` —which both return `true` only when `withdrawable_at == 0` . These methods perform the same check, resulting in redundant logic and potential confusion about whether unstaking and claiming rewards have different eligibility conditions. Maintaining duplicate logic increases the risk of future inconsistencies if one method is updated independently.

```
impl StakeReceipt {
    ...

    /// Returns whether or not the StakeReceipt is in a state that allows
unstaking.
    pub fn can_unstake(&self) -> bool {
        self.withdrawable_at == 0
    }

    /// Returns whether or not the StakeReceipt is in a state that allows
claiming rewards.
    pub fn can_claim_rewards(&self) -> bool {
        self.withdrawable_at == 0
    }
```

Recommendation

It is recommended to consolidate these functions into a single method (e.g., `is_active()` or `is_stake_locked()`) that expresses the underlying condition clearly. If future differentiation is needed, the method can be extended, but until then, reducing duplication improves clarity and maintainability.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	stake_pool.rs#365
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
// Transfer the reward tokens
let cpi_accounts = Transfer {
    from: reward_vault_info.clone(),
    to: user_reward_token.clone(),
    authority: stake_pool_account.clone(),
};
let cpi_ctx = CpiContext {
    accounts: cpi_accounts,
    remaining_accounts: vec![],
    program: token_program_info.clone(),
    signer_seeds: &[stake_pool_signer_seeds!(self)],
};
token::transfer(cpi_ctx, total_claimable)?;
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

UVP - Unchecked Vault Parameters

Criticality	Minor / Informative
Location	stake.rs#29
Status	Unresolved

Description

The contract does not perform runtime checks to validate critical properties of the `vault` account, such as confirming that the `mint` matches the expected token and that the `owner` is the `stake_pool`. Without these checks, it is possible for a malicious or misconfigured vault account to be passed to the instruction, leading to incorrect accounting, misrouted funds, or unauthorised control over token balances. Relying solely on account constraints at the macro level does not guarantee correctness unless all assumptions are explicitly validated at runtime.

```
    #[account(mut)]
     pub vault: Account<'info, TokenAccount>,
```

Recommendation

It is recommended to include runtime validation logic that ensures the `vault.mint` matches the expected token mint and that the `vault.owner` is set to the `stake_pool` address. These checks should be added early in the instruction handler or `validate` method to prevent improper vault associations and protect the integrity of token operations.

USU - Unnecessary Struct Usage

Criticality	Minor / Informative
Location	create_stake_pool.rs#36 stake.rs#84
Status	Unresolved

Description

The contract is using a struct, `CreateStakePoolArgs`, to encapsulate a single field, `authority`, which serves as the only parameter in the context of stake pool creation. While this approach may be anticipating future extensibility, it introduces unnecessary abstraction and overhead for the current implementation. The presence of a struct for a single value complicates the interface, potentially misleading future maintainers into believing there is or will be multiple parameters involved. This design choice reduces clarity without offering functional benefits in its current form.

```
pub struct CreateStakePoolArgs {
    /// The key that will be the authority over the StakePool
    pub authority: Pubkey,
}
```

Recommendation

It is recommended to replace the struct with a standalone variable, especially since only one parameter is used. This simplifies the function signature, improves readability, and avoids implying unnecessary complexity. If additional parameters are expected in future versions, the struct can be reintroduced at that time.

UAI - Unvalidated Authority Input

Criticality	Minor / Informative
Location	create_stake_pool.rs#36 update_authority.rs29
Status	Unresolved

Description

The contract accepts a `Pubkey` as input to assign or update authority roles—such as in stake pool creation or authority updates—without validating that the provided key is a valid and non-default address. This pattern appears in multiple instruction argument structs (e.g., `CreateStakePoolArgs`, `UpdateAuthorityArgs`) where the input authority can be set to `Pubkey::default()` (i.e., the all-zero address). This omission may lead to critical misconfigurations, such as unintentionally assigning control to an unusable or unowned address, ultimately resulting in permanent loss of administrative access or the inability to manage protocol operations.

```
pub struct CreateStakePoolArgs {
    /// The key that will be the authority over the StakePool
    pub authority: Pubkey,
}
```

```
#[derive(AnchorDeserialize, AnchorSerialize)]
pub struct UpdateAuthorityArgs {
    pub new_authority: Pubkey,
}

pub fn handler(ctx: Context<UpdateAuthority>, args: UpdateAuthorityArgs)
-> Result<()> {
    let stake_pool = &mut ctx.accounts.stake_pool;
    stake_pool.authority = args.new_authority;

    Ok(())
}
```

Recommendation

It is recommended to implement validation logic to ensure that any authority-related input is not equal to the default public key. This check should be enforced in the corresponding `validate` functions or directly within handler logic to ensure proper contract configuration and to prevent accidental or malicious assignment of invalid authority values.

Summary

Mage Labs contract implements a weighted staking and reward distribution mechanism supporting both fungible tokens and NFTs. This audit investigates security issues, business logic concerns, and potential improvements to ensure correctness, efficiency, and readiness for production deployment.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

A **TAC Security** Company

The Cyberscope team

cyberscope.io