

Solomon Labs

Security Review

Cantina Managed review by:
Mustafa Hasan, Lead Security Researcher
Mario Poneder, Security Researcher
Security Review Advisor: **Zigtur**

January 25, 2026

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
- 2.1 Scope . . . 3
- 2.2 Previously-known issues . . . 3
  - 2.2.1 Vault program . . . 3
  - 2.2.2 Stake program . . . 4
- 2.3 Architecture review . . . 5
  - 2.3.1 Vault and Stake as a system . . . 5
  - 2.3.2 Design choices that might hurt later . . . 6
- 2.4 Future-facing considerations . . . 7
  - 2.4.1 Token-2022 and a dedicated locking/boost program . . . 7

3 Findings **9**
- 3.1 Medium Risk . . . 9
  - 3.1.1 Redemption rate can be set to zero without restriction leading to stuck redemptions . . . 9
  - 3.1.2 `min_shares` invariant is never enforced in staking/unstaking flows . . . 9
  - 3.1.3 Incorrect `space` calculations for vault/stake state accounts . . . 10
  - 3.1.4 Rewarding into an empty vault permanently bricks staking . . . 12
- 3.2 Low Risk . . . 13
  - 3.2.1 Unused error variants in Vault and Stake obscure actual guarantees . . . 13
  - 3.2.2 Vesting period updates mid-vesting may lead to DoS . . . 13
  - 3.2.3 Redeemers can grief rewarding logic . . . 13
  - 3.2.4 `UpdateAsset` does not bind `asset` to `collateral_token_mint` weakening the 9-decimal safety invariant . . . 14
  - 3.2.5 Unstake queue length limit is not enforced . . . 15
  - 3.2.6 Off-by-one cap checks allow `MAX + 1` managers/withdraw addresses . . . 15
  - 3.2.7 Admin changing `vesting_period` can break `total_assets >= get_unvested()` invariant and cause panics in share math . . . 16
- 3.3 Informational . . . 17
  - 3.3.1 `Unstake` account context stack frame exceeds 4 KiB limit causing access violations . . . 17
  - 3.3.2 Single-step admin transfer may lead to DoS . . . 18
  - 3.3.3 Caller pubkey not emitted in `NewVaultEvent` events . . . 18
  - 3.3.4 Uncapped cooldown period . . . 18
  - 3.3.5 Rewarders vector can be empty . . . 19
  - 3.3.6 Possible overflow in value casting . . . 19
  - 3.3.7 Zero `redeem_rate` can disable redemptions while deposits remain enabled . . . 19

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Jan 13th to Jan 20th the Cantina team conducted a review of solana-programs-audit-r2 on commit hash 17a589e7. The team identified a total of **18** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	4	0	4
Low Risk	7	0	7
Gas Optimizations	0	0	0
Informational	7	0	7
Total	**18**	**0**	**18**

2.1 Scope

The security review had the following components in scope for solana-programs-audit-r2 on commit hash 17a589e7:

```
programs
├── stake
│   ├── Cargo.toml
│   └── src
│       ├── context.rs
│       └── lib.rs
└── vault
    ├── Cargo.toml
    ├── src
    │   ├── context.rs
    │   └── lib.rs
    └── Xargo.toml
```

2.2 Previously-known issues

2.2.1 Vault program

- **High: Use of deposit rate instead of redeem rate in `redeem` - fixed:**
  - Spec: `redeem` must use `redeem_rate` (asset units per stablecoin), not `deposit_rate`.
  - Assessment: The redeem path no longer uses `deposit_rate`; this issue is resolved as described.
- **Medium: Unsafe `u128` → `u64` casts - fixed:**
  - Spec: Replace unsafe `as` casts with checked conversions to prevent silent overflow.
  - Assessment: The unsafe `as` casts are gone, replaced by `try_into()`, so arithmetic overflow will panic instead of silently wrapping (which is an improvement).
- **Medium: Inconsistent rate scaling - fixed:**
  - Spec: Both deposit and redeem must use the same 9-decimals scaling, and reject collateral mints with more than 9 decimals.
  - Assessment: Fixed as specified; scaling logic is consistent and mints with decimals > 9 are rejected.
- **Low/Info: Manager and withdrawal address caps not enforced - partially fixed:**
  - Spec: Enforce max 20 `asset_managers`, 20 `role_managers`, 50 `withdraw_addresses`.

- Assessment: The absence of any cap is fixed, but the implementation is off by one and slightly mis-sized. This is a low-severity correctness/operational risk, not an exploitable bug, but it does not perfectly match the ”caps are enforced” claim.

- **Low/Info: Events lacked key fields - fixed:**
  - Spec: Include token mint, old/new rates, manager type, etc.
  - Assessment: Fixed; events contain the necessary IDs and amounts for effective observability / auditability.
- **Low/Info: Misleading ATA comments - fixed:**
  - Spec: The comments on `program_collateral` and `caller_*` accounts should accurately describe them.
  - Assessment: The comments now correctly describe them as regular token accounts, not ATAs.
- **Info: Initialization front-running and admin centralization - unchanged, by design:**
  - Spec: The initializer picks the admin, who can arbitrarily set managers, withdrawal addresses, and rates.
  - Assessment:
    * No access control on `initialize_vault_state`; the initializer picks the `admin`, who can then arbitrarily set managers, withdrawal addresses, and rates (including to drain or freeze the vault). Manageable risk with robust deployment, initialization and an admin sanity check after initialization.
    * These are unchanged and match the acknowledged trade-offs. They are acceptable only if the admin is a robust, well-governed entity; they should remain clearly documented as centralization risks.

2.2.2 Stake program

- **Medium: `initialize_program_accounts` lacked access control and deposit-token consistency - fixed:**
  - Spec: Only the vault admin should be able to call it; the supplied `deposit_token` must match `VaultState.deposit_token`.
  - Assessment: Fixed as specified.
- **Medium: No way to un-blacklist - fixed:**
  - Spec: Add an instruction to remove users from the blacklist.
  - Assessment: Fixed, with symmetric add/remove behaviour guarded by admin-only access.
- **Medium: Reward vesting could be bypassed by overlapping `reward` calls - fixed:**
  - Spec: Prevent a rewarder from resetting `last_distribution_time` and `vesting_amount` while a previous vesting period is still in progress.
  - Assessment: The specific ”overlapping reward calls skip vesting” issue is fixed.
- **Medium: Unsafe `u128` → `u64` casts - fixed:**
  - Spec: Replace unsafe casts with checked conversions to avoid silent overflow.
  - Assessment: Uses `u128` intermediates and `try_into()`, so no silent wrap; this matches the ”use checked conversions” intent.
- **Low: Events lacked vault identification - fixed:**
  - Spec: Events must include a vault identifier (e.g. `salt`).
  - Assessment: Fixed.
- **Low: Cooldown decrease could leave users waiting too long - partially fixed:**
  - Spec: Add a mechanism so users can benefit from shortened cooldowns on existing queue entries.

  - Assessment: Behaviour matches the ”partially fixed” description:
    * Users with existing entries can call `refresh_cooldowns` to bring their unlock times forward after a cooldown decrease.
    * The code never extends existing unlock times when cooldown is increased.
- **Low: `get_unvested` divide-by-zero - fixed:**
  - Spec: Avoid division by zero in `get_unvested`.
  - Assessment: The original ”vesting period 0” divide-by-zero in `get_unvested` is fixed.
- **Info / acknowledged: front-runnable `initialize_venture_state` via `salt` - unchanged, by design:**
  - Spec: The permission structure allows front-running via salt selection.
  - Assessment: Still front-runnable, as documented. Given the large salt space and the ability to choose a new salt, this is an acceptable, explicitly acknowledged trade-off rather than a hidden vulnerability.
- **Info: Rewarders vector bound not enforced - unchanged:**
  - Spec: The account is sized for 20 `rewarders` but the logic does not enforce that cap.
  - Assessment: Exactly as described: the account is sized for 20 `rewarders` but the logic does not enforce that cap.
    * Primary risk is state/serialization failure.
    * Only the admin can add/remove rewarders, so this is an operational footgun, not an untrusted-user exploit. It remains an acknowledged limitation.
- **Info: Slight over-allocation of `vault_state` space - partially fixed, new sizing issue:**
  - Spec: Original issue was ”slightly over-allocated `vault_state` space”.
  - Assessment: The over-allocation is no longer present; instead, the account is now slightly under-allocated (missing the 4-byte `Vec` length field for `rewarders`). This is a new correctness/DoS risk (potential serialization failure near capacity).
- **Info: Minor efficiency concerns in blacklist PDA - unchanged:**
  - Spec: The blacklist PDA structure results in extra rent and per-user storage.
  - Assessment: Matches the acknowledged design; the main impact is extra rent and storage per user, not a correctness/security problem.

2.3 Architecture review

2.3.1 Vault and Stake as a system

- Vault program is a governed, whitelisted mint/redeem bridge between external collateral mints and a single protocol stablecoin. All solvency and price assumptions flow through:
  - The admin's configuration of exchange rates per collateral mint, and.
  - The role/withdraw-address configuration, which is effectively an on-chain ACL on who can mint/redeem and who can move collateral out to custody.
- Stake program is a multi-vault, ERC-4626-style staking layer on top of arbitrary deposit tokens (including the Vault's stablecoin), with:
  - Time-based vesting of rewards,
  - Cooldown queues for exits,
  - A per-vault admin that controls rewarders and blacklist,
  - And no coupling back into the Vault program by code (only via which mint you choose as `deposit_token`).

As a system, the dominant pattern is: Vault defines the monetary asset and its admin power; Stake defines how that asset is locked and rewarded under a separate, per-vault admin. There is no circular dependency, which is good for upgradability.

2.3.2 Design choices that might hurt later

- **Vault:**
  - Single global `VaultState` + fully trusted admin.
    - All collateral, minters/redeemers, managers, and withdrawal addresses are recorded under one `VaultState` controlled by one admin.
    - Why it hurts later: any future desire to have ”semi-independent” vaults (e.g. per-asset vaults with different admins/risk policies for collateral assets) will require a breaking change to the PDA layout or a v2 program. Today's layout bakes in ”one global admin, many collateral assets” very hard.
  - Rates as freeform knobs, no safety rails.
    - Admin can set `deposit_rate` and `redeem_rate` arbitrarily, and can set `redeem_rate = 0` while `deposit_rate > 0` and/or funds still remaining to redeem.
    - Why it hurts later: even if you trust governance, it's easy to misconfigure assets. If you ever want stronger UX guarantees (”this asset cannot be configured into a stuck state”), you'll need on-chain constraints and probably a two-step, timelocked change flow.
  - Per-user permissions PDAs keyed only by user, not by asset.
    - `Permissions` are per `(vault_state, user)`, not per asset, yet they gate mint/redeem for all collateral.
    - Why it hurts later: if you want asset-scoped permissions (e.g. user can mint only against USDC but not USDT), you can't express that without changing the PDA layout or adding more indirection.
- **Stake:**
  - Mixing ERC-4626 math, cooldown queues, rewards, and blacklist in one program.
    - The stake program does everything: share pricing, reward vesting, queue management, admin/rewarder config, blacklist.
    - Why it hurts later: any attempt to:
      - Change queue structure/rent model,
      - Add ”boost” logic,
      - Or migrate to Token-2022-style locking. will risk destabilizing the core ERC-4626 semantics. Splitting into:
      - A small, stable ”pure vault” program, and.
      - A more flexible policy/locking program. would make future evolution much safer.
  - Reward semantics tied tightly to time and `vesting_period`.
    - `convert_to_shares/convert_to_assets` depend on `get_unvested()` and the current `vesting_period`, and changes to `vesting_period` can affect share math mid-stream.
    - Why it hurts later: it couples configuration changes (admin changing vesting) directly into pricing logic. As you add more complex reward policies (e.g. multi-reward tokens, phase-based vesting), this coupling will become brittle and bug-prone unless you isolate ”how much is vested” from ”how shares are priced”.
  - Per-address unstake queues as big, independent PDAs.
    - Every active address gets a relatively heavy `UserPDA` with a queue and its own rent cost.
    - Why it hurts later: at protocol scale, the combination of:
      - One PDA per address per vault,

        - Straightforward `VecDeque` queue layout, and.
        - No shared batching. will show up as high UX costs and noisy DoS edges (e.g. accounts filling queues and hitting size limits).

- **System-level:**
  - Trust boundary is very thin: ”admin + rewarders” vs everyone else.
    - Right now, almost all nontrivial power lives with admins/rewarders; everyone else is constrained by static math and PDAs.
    - Why it hurts later: if governance ever wants to delegate some responsibilities (e.g. asset-specific managers, different admins per staking vault using the same asset), the current ”single super-admin per program/per salt” architecture doesn't give you much granularity. You'll likely need:
        - Either a governance layer that programs in constraints off-chain (timelock, policies), or.
        - A v2 that encodes richer role separation directly on-chain.

Overall, the code is comparatively clean for the current feature set, but the strong central admin model, the tight coupling between configuration and math, and the feature-rich Stake design are the main things that will make future extensions (per-asset governance, complex rewards/boosts, cheaper queues) troublesome if not planned for explicitly.

2.4 Future-facing considerations

2.4.1 Token-2022 and a dedicated locking/boost program

Token-2022 migration:
Both programs currently assume plain SPL Token mints, meaning no transfer hooks or on-chain fees are modeled. If you migrate to Token-2022, you must:

- Add explicit checks in `update_asset` to whitelist or reject unsupported extensions (confiscation, transfer hooks, interest/fees) on collateral mints, so you don't silently inherit behaviors that aren't modeled.
- For fee-bearing collateral mints, you must account on net amounts (what the vault actually receives/sends) or you risk hidden, accumulating insolvency as transfer fees are taken.
- In Stake, because the staking token mint is created by your program, you can fully control its extensions; the bigger risk is adoption of Token-2022 by the external `deposit_token` mint. There, you should inspect/whitelist extensions, or consider pairing with a dedicated locking/boost program that uses Token-2022 transfer hooks.

Dedicated locking/boost program:
For future ”boosts” (e.g. longer locks $\rightarrow$ higher rewards), it is safer to:

- Keep the current Stake program as a thin, core ERC-4626 vault,
- Move boosts/lockups into a separate program that escrows staking tokens and tracks multipliers. This isolates complex incentive logic from core share math and makes upgrades less risky.

Vanity addresses and separation of concerns:
Deployments may want human-readable program/mint addresses (e.g. vanity prefixes) via address search at deploy. Your PDA schemes (e.g. `VAULT_STATE_SEED`, `STAKING_TOKEN_SEED`) are already deterministic, meaning vanity IDs are safe if all clients use the same derivation rules. Just ensure strong tooling and documentation so vanity remains a deployment detail—never changing on-chain logic.

Stake currently couples core (ERC-4626 math, staking) with policy layers (cooldown queues, blacklist, rewards, admin config). For long-term maintainability, consider splitting these:

- A minimal ”core vault” program (stake/unstake/reward and math only),
- A separate policy/extension layer (cooldown, vesting, blacklist, additional admin logic). This makes future changes (such as boosts or policy upgrades) easier and safer.

Reducing blacklist and queue costs:
Today, every address that touches Stake pays for its own blacklist PDA, even though most will never be blacklisted.

- Instead, only create the blacklist PDA when a user is actually blacklisted (during the `blacklist` instruction).
- The PDA's presence means the account is blacklisted; by default, absence means not blacklisted.
- To remove from blacklist, the admin simply closes (reclaims rent on) the PDA, instead of toggling a flag.
  This approach drastically reduces rent costs for legitimate users.

Unstake queue compression:
There's no clean way to compress the unstake queue meaningfully without changing semantics. The best you can do (if desperate) is coarse adjustments like:

- Bucketing unlock times into days or weeks (not exact timestamps),
- Reducing amount precision (e.g. round to nearest cent).
  Anything more complex adds risk of bugs for little rent savings, so only consider these if rent costs ever become a serious bottleneck.

Tightening admin invariants:
Currently, `deposit_rate`/`redeem_rate` are unconstrained. For safety, model each vault/asset lifecycle explicitly:

- Active: `deposit_rate > 0 && redeem_rate > 0` (normal ops)
- WindDown: `deposit_rate = 0 && redeem_rate > 0` (no inflows, exits still allowed)
- Frozen: `deposit_rate = 0 && redeem_rate = 0` (no inflows or outflows; only governance ops)
  Require strict state transitions (e.g. only Active $\rightarrow$ WindDown $\rightarrow$ Frozen, not skipping steps). Emit explicit events on mode changes so off-chain watchers see when an asset is being wound down.

Two-step flow for high-impact changes:
For risky changes (e.g. setting `redeem_rate` to 0, major cooldown/vesting changes, mode to Frozen, admin transfer):

- Require a `propose_change` call that logs the new value and emits an intent event,
- Require a later `execute_change` call after a delay (block/slot-based).
  This is compatible with multisig + timelock, and gives off-chain actors time to react before changes take effect.

Separate "config admin" from "ops/admin helpers":
Powers should be split into at least two roles:

- Config admin: can change economic parameters (rates, modes, cooldown, vesting); protected by timelock/governance.
- Operational managers: may add/remove rewarders, withdrawal addresses, or role managers, but have zero access to rates, modes, or other monetary invariants.
  This reduces the blast radius of operational errors and clarifies which actors can brick or reconfigure the system.

3 Findings

3.1 Medium Risk

3.1.1 Redemption rate can be set to zero without restriction leading to stuck redemptions

Severity: Medium Risk

Context: lib.rs#L111

Description: `redeem_rate` can be set to zero at any point in time, leading to minters being unable to redeem for that particular mint.

Recommendation: Emit an event and set a flag to signal redemptions are paused for the asset.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.1.2 `min_shares` invariant is never enforced in staking/unstaking flows

Severity: Medium Risk

Context: context.rs#L163-L170, context.rs#L279-L286, lib.rs#L47, lib.rs#L73

Description: The `VaultState` includes a `min_shares` parameter and there are helper methods intended to enforce it, but these helpers are never called, so the invariant is effectively never enforced.

- State field:

```
pub struct VaultState {
    pub admin: Pubkey,
    pub deposit_token: Pubkey,
    pub vesting_amount: u64,
    pub total_assets: u64,
    pub min_shares: u64,
    // ...
}
```

- Intended checks on stake/unstake contexts:

```
impl<'info> Stake<'info> {
    pub fn check_min_shares(&self) -> Result<()> {
        let shares = self.staking_token.supply;
        if self.user.key() != self.vault_state.admin && shares > 0 && shares <
        ↪  self.vault_state.min_shares {
            return Err(StakeError::MinSharesViolation.into())
        }
        Ok(())
    }
}
```

```
impl<'info> Unstake<'info> {
    pub fn check_min_shares(&self) -> Result<()> {
        let shares = self.staking_token.supply;
        if self.user.key() != self.vault_state.admin && shares > 0 && shares <
        ↪  self.vault_state.min_shares {
            return Err(StakeError::MinSharesViolation.into())
        }
        Ok(())
    }
}
```

- But these functions are never invoked in the staking logic.

As a result, non-admin users can freely operate in regimes where `staking_token.supply` is below `min_shares`, defeating any intended protection against very low-liquidity / low-share-count states and contradicting what the presence of `min_shares` and `StakeError::MinSharesViolation` suggest.

Recommendation: It is recommended to revise the intended semantics for `min_shares` and either enforce the invariant or remove the associated code.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.1.3 Incorrect space calculations for vault/stake state accounts

Severity: Medium Risk

Context: context.rs#L9-L16, context.rs#L224-L231, context.rs#L425-L432, context.rs#L14-L21

Description: Three accounts are under-allocated relative to their structs and intended capacities:

- Stake - `VaultState` (`InitializeVaultState`).

```
#[account(
    init,
    payer = caller,
    space = 8 + (2 * 32) + (3 * 8) + (3 * 4) + 1 + (32 * 20),
    seeds = [VAULT_STATE_SEED, salt.as_ref()],
    bump
)]
pub vault_state: Box<Account<'info, VaultState>>,
```

```
pub struct VaultState {
    pub admin: Pubkey,              // 32
    pub deposit_token: Pubkey,      // 32
    pub vesting_amount: u64,        // 8
    pub total_assets: u64,          // 8
    pub min_shares: u64,            // 8
    pub last_distribution_time: u32,// 4
    pub cooldown: u32,              // 4
    pub vesting_period: u32,        // 4
    pub bump: u8,                   // 1
    pub rewarders: Vec<Pubkey>,     // 4 + 20 * 32
}
```

  - Required struct size (for 20 `rewarders`):
    - Fixed fields: `2*32 + 3*8 + 3*4 + 1 = 101`.
    - `rewarders`: `4 + 20*32 = 644`.
    - Struct: `101 + 644 = 745`.
    - With discriminator: `8 + 745 = 753` bytes.
  - Current `space`: `749` bytes → 4 bytes too small (missing the 4-byte `Vec` length).
- Stake - `UserPDA` (`Unstake` / `RefreshCooldowns`).

```
#[account(
    init_if_needed,
    payer = user,
    space = 8 + 8 + 100 * 12,
    seeds = [USER_DATA_SEED, user.key().as_ref(), vault_state.key().as_ref()],
    bump
)]
pub user_data: Account<'info, UserPDA>,
```

```
pub struct UserPDA {
    pub assets_available: u64,              // 8
    pub unstake_queue: VecDeque<(u32, u64)>,// 4 + 100 * 12
}
```

  - Element size `(u32, u64)`: `4 + 8 = 12`.
  - Required struct size (for 100 entries):

    - Fixed fields: `8`.
    - `unstake_queue`: `4 + 100*12 = 1204`.
    - Struct: `8 + 1204 = 1212`.
    - With discriminator: `8 + 1212 = 1220` bytes.
  - Current `space`: `8 + 8 + 100*12 = 1216` bytes → 4 bytes too small (missing the 4-byte `VecDeque` length).
- Vault - `VaultState` (`InitializeVaultState`).

```
#[account(
    init,
    payer = signer,
    space = 8 + 8 + (2 + 20 + 20 + 50) * 32,
    seeds = [VAULT_STATE_SEED],
    bump
)]
pub vault_state: Account<'info, VaultState>,
```

```
pub struct VaultState {
    pub vault_token_mint: Pubkey,        // 32
    pub asset_managers: Vec<Pubkey>,     // 4 + 20 * 32
    pub role_managers: Vec<Pubkey>,      // 4 + 20 * 32
    pub withdraw_addresses: Vec<Pubkey>,// 4 + 50 * 32
    pub admin: Pubkey,                   // 32
    pub bump: u8,                        // 1
}
```

  - Required struct size (20/20/50 capacities):
    - Fixed fields: `32 + 32 + 1 = 65`.
    - `asset_managers`: `4 + 20*32 = 644`.
    - `role_managers`: `4 + 20*32 = 644`.
    - `withdraw_addresses`: `4 + 50*32 = 1604`.
    - Vecs: `644 + 644 + 1604 = 2892`.
    - Struct: `65 + 2892 = 2957`.
    - With discriminator: `8 + 2957 = 2965` bytes.
  - Current `space`: `8 + 8 + (2 + 20 + 20 + 50)*32 = 2960` bytes → 5 bytes too small (bump and three `Vec` lengths not fully covered by the lump `+ 8`).

In all three cases, accounts are slightly undersized; as dynamic collections approach their intended maximum lengths, serialization/deserialization can panic and brick the relevant instructions.

Recommendation: It is recommended to consider the following mitigation measures:

- For each account, set `space` to:
  - `8` (discriminator).
  - Plus the sum of:
    - Fixed field sizes, and...
    - For each `Vec<T>` / `VecDeque<T>`: `4 + max_len * size_of::<T>()`.
- Concretely:
  - Stake `VaultState`:

```
    space = 8 + (2 * 32) + (3 * 8) + (3 * 4) + 1 + 4 + (32 * 20); // 753
```

  - Stake `UserPDA`:

```
    space = 8 + 8 + 4 + 100 * 12; // 1220
```

- Vault `VaultState`:

```
space = 8 + 32 + 32 + 1 + (4 + 20 * 32) + (4 + 20 * 32) + (4 + 50 * 32); //
↪  2965
```

Solomon Labs: Acknowledged. The admin-related fields wont be a problem as an operational issue, the stake pda could in practice although unlikely.

Cantina Managed: Acknowledged.

3.1.4 Rewarding into an empty vault permanently bricks staking

Severity: Medium Risk

Context: lib.rs#L267-L273, lib.rs#L368-L370, lib.rs#L419-L428

Description: When a reward is distributed before any shares exist (i.e. `staking_token.supply == 0`), the vault is effectively bricked and no subsequent user can ever stake. Flow:

- Initial state: `vault_state.total_assets == 0`, `staking_token.supply == 0`.
- A rewarder calls `reward`, which sets:

```
// via reward()
state.total_assets += amt;
state.vesting_amount = amt;
state.last_distribution_time = time;
```

- Now `total_assets > 0`, but `staking_token.supply == 0`.
- On the first `stake`, shares are computed as:

```
pub fn convert_to_shares(&self, assets: u64, total_supply: u64) -> Result<u64>  {
    if self.total_assets == 0 {
        return Ok(assets);
    }

    let total_assets = self.total_assets - self.get_unvested()?;
    let x = (assets as u128 * total_supply as u128 / total_assets as
    ↪  u128).try_into().unwrap();
    Ok(x)
}
```

- Because `self.total_assets != 0` and `total_supply == 0`, `x` is computed from `0 / total_assets`, which yields `0` shares; `stake` then returns `StakeError::ZeroShares` for all users and amounts:

```
let shares = ctx.accounts.vault_state.convert_to_shares(
    amt,
    ctx.accounts.staking_token.supply,
)?;
if shares == 0 {
   return Err(StakeError::ZeroShares.into())
}
```

There is no path to recover from this state (short of an upgrade), so a single reward into an empty vault permanently prevents any future staking.

Recommendation: It is recommended to forbid rewards when there are no shares, e.g.:

```
pub fn reward(ctx: Context<Reward>, amt: u64, salt: [u8; 8]) -> Result<()> {
    require!(ctx.accounts.staking_token.supply > 0, StakeError::ZeroShares);
    // existing logic...
}
```

Solomon Labs: Acknowledged. In practice this wont happen, but we note it down.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 Unused error variants in Vault and Stake obscure actual guarantees

Severity: Low Risk

Context: lib.rs#L590-L596, lib.rs#L632-L659

Description: Both programs define error variants that are never emitted in the current code. Some are harmless leftovers; a few suggest features or checks that don't exist.

Stake program (`StakeError`):

- `UserNotFound`. No code searches a global user list; all flows are driven by caller PDAs, so this error is not needed under the current design.
- `WrongToken`. Anchor `token::mint` constraints enforce token correctness; this error is not needed.
- `BadStakingTokenDecimals`. Staking mint decimals are always set equal to `deposit_token.decimals` in `initialize_program_accounts`, so no runtime mismatch check is needed.

Vault program (`VaultError`):

- `AlreadyMinter`. `whitelist_minter` is effectively idempotent; a dedicated ”already” error is currently unnecessary unless stricter behaviour is desired.
- `AlreadyRedeemer`. Same as `AlreadyMinter`; adds no value without a corresponding check.
- `MinterNotWhitelisted` / `RedeemerNotWhitelisted` / `AddressNotWhitelisted`. The code uses `NotAnApprovedMinter`, `NotAnApprovedRedeemer`, and `NotWithdrawer` instead; these extra variants are redundant.
- `MaxMintExceeded` / `MaxRedeemExceeded`. There is no per-block/epoch mint or redeem limit implemented; these errors hint at a feature that does not exist.
- `AssetAlreadySupported`. `update_asset` allows updating an existing asset in place; there is no ”first-time only” restriction today.

Recommendation: It is recommended to remove clearly redundant errors so the enums reflect only the actually used cases and implement the corresponding checks in case of ”feature” errors, if intended.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.2 Vesting period updates mid-vesting may lead to DoS

Severity: Low Risk

Context: lib.rs#L148

Description: The `set_vesting_period()` function can be called at any point in time, even if vesting is already in progress. This may lead to a DoS case by stretching the vesting period to a long one.

Recommendation: It would make sense if this function is only strictly callable before vesting commences.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.3 Redeemers can grief rewarding logic

Severity: Low Risk

Context: lib.rs#L366

Description: The `reward()` function can be called by rewarders at any point in time to accumulate rewards. However, the `amt` value passed to the function can be zero and the token transfer CPI will not revert in such a case. This allows rewarders to keep pushing vesting schedules without actual reward accumulation, leading to reward griefing on the users' side.

Recommendation: Revert if the passed value is below a certain threshold for a reward to be considered.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.4 `UpdateAsset` does not bind `asset` to `collateral_token_mint` weakening the 9-decimal safety invariant

Severity: Low Risk

Context: context.rs#L47-L74, lib.rs#L121-L124

Description: The Vault's `update_asset` instruction is intended to enforce that only collateral mints with $\leq$ 9 decimals are supported. However, the implementation allows registering an `asset` key whose decimals were never actually checked.

Relevant code:

```
// Context
#[derive(Accounts)]
#[instruction(asset: Pubkey)]
pub struct UpdateAsset<'info> {
    #[account(
        init_if_needed,
        payer = authority,
        seeds = [EXCHANGE_RATE_SEED, asset.as_ref()],
        space = 8 + 8 + 8 + 32,
        bump
    )]
    pub exchange_rate: Account<'info, ExchangeRate>,

    /// The program owned collateral
    #[account(
        init_if_needed,
        seeds = [TOKEN_ACCOUNT_SEED, asset.as_ref()],
        bump,
        payer = authority,
        token::mint = collateral_token_mint,
        token::authority = vault_state,
    )]
    pub program_collateral: Account<'info, TokenAccount>,

    #[account(mut)]
    pub collateral_token_mint: Account<'info, Mint>, // no equality check with `asset`
    // ...
}
```

```
// Handler
pub fn update_asset(
    ctx: Context<UpdateAsset>,
    asset: Pubkey,
    deposit_rate: u64,
    redeem_rate: u64,
) -> Result<()> {
    if ctx.accounts.authority.key() != ctx.accounts.vault_state.admin {
        return Err(VaultError::NotAdmin.into());
    }

    // Decimals check is performed on collateral_token_mint, not on `asset`
    if ctx.accounts.collateral_token_mint.decimals > 9 {
        return Err(VaultError::AssetTooManyDecimals.into());
    }

    ctx.accounts.exchange_rate.asset = asset;
    ctx.accounts.exchange_rate.deposit_rate = deposit_rate;
    ctx.accounts.exchange_rate.redeem_rate = redeem_rate;
    // ...
}
```

`ExchangeRate` and the program-owned collateral account are keyed by `asset` (via `asset.as_ref()` in the seeds), but `asset` is never required to equal `collateral_token_mint.key()`. If an admin mistakenly passes an `asset` that does not match the provided mint:

- The program will create/update PDAs for a logical "asset" whose decimals were never validated.
- `exchange_rate.asset` (emitted via `AssetModifiedEvent.asset`) does not correspond to the actual SPL mint of the collateral, confusing off-chain monitoring and governance.
- Deposits/redeems for `collateral_token_mint` will continue to use PDAs keyed by `collateral_token_mint.key()`, ignoring the mis-keyed `exchange_rate`, so the vault's internal state and monitoring data can diverge from reality.

This undermines a documented safety invariant and increases the risk of misconfiguration and mis-accounting.

Recommendation: It is recommended to remove the `asset` argument and field entirely, and always use `collateral_token_mint.key()` as the seed and stored `asset` identifier. This eliminates the possibility of divergence between the logical asset key and the mint whose decimals were validated.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.5 Unstake queue length limit is not enforced

Severity: Low Risk

Context: lib.rs#L304

Description: `UserPDA` is sized for a queue of 100 `(u32, u64)` entries, but `start_unstake` never checks the queue length before pushing a new item:

```
// fn start_unstake
ctx.accounts.user_data.unstake_queue.push_back((cd_end, assets)); // no length check
```

As a result, users can accumulate more than 100 entries in `unstake_queue`, exceeding the allocated space, leading to serialization panics instead of a clean error.

Recommendation: It is recommended to add an explicit length check in `start_unstake`, e.g.:

```
const MAX_QUEUE_LEN: usize = 100; // keep in sync with space calculation for UserPDA
let queue = &mut ctx.accounts.user_data.unstake_queue;
if queue.len() >= MAX_QUEUE_LEN {
    return Err(StakeError::UnstakeQueueFull.into()); // define a dedicated error
}
queue.push_back((cd_end, assets));
```

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.6 Off-by-one cap checks allow `MAX + 1` managers/withdraw addresses

Severity: Low Risk

Context: lib.rs#L359-L369, lib.rs#L386-L396, lib.rs#L455-L465

Description: The Vault program enforces maximum counts for asset managers, role managers, and withdrawal addresses, but the checks are off by one. They use `len() > MAX` instead of `len() >= MAX`, which allows `MAX + 1` entries:

```
// fn add_asset_manager
let managers = &mut ctx.accounts.vault_state.asset_managers;
if managers.len() > MAX_MANAGER_ADDRESSES {
    return Err(VaultError::MaxArrayLength.into());
}
```

```rust
// fn add_role_manager
let managers = &mut ctx.accounts.vault_state.role_managers;
if managers.len() > MAX_MANAGER_ADDRESSES {
    return Err(VaultError::MaxArrayLength.into());
}
```

```rust
// fn add_withdraw_address
let withdraw_addresses = &mut ctx.accounts.vault_state.withdraw_addresses;
if withdraw_addresses.len() > MAX_WITHDRAW_ADRESSES {
    return Err(VaultError::MaxArrayLength.into());
}
```

Given `MAX_MANAGER_ADDRESSES = 20` and `MAX_WITHDRAW_ADRESSES = 50`, the current logic permits 21 managers or 51 withdrawal addresses before reverting. This contradicts the documented caps and leads to serialization failures when arrays are filled above their intended maximum sizes instead of triggering the above errors.

Recommendation: It is recommended for all three checks to change the condition from `len() > MAX_...` to `len() >= MAX_...` so that:

- The maximum number of entries is exactly `MAX_...` and...
- Attempts to add the `MAX_... + 1`-th entry correctly returns `VaultError::MaxArrayLength`.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.2.7 Admin changing `vesting_period` can break `total_assets >= get_unvested()` invariant and cause panics in share math

Severity: Low Risk

Context: lib.rs#L419-L439, lib.rs#L441-L453

Description: In `VaultState::convert_to_shares` and `VaultState::convert_to_assets`, the program assumes `self.total_assets >= self.get_unvested()?` and computes "vested assets" as:

```rust
pub fn convert_to_shares(&self, assets: u64, total_supply: u64) -> Result<u64>  {
    if self.total_assets == 0 {
        return Ok(assets);
    }

    let total_assets = self.total_assets - self.get_unvested()?; // “vested assets”:
    ↪  assumes no underflow
    let x = (assets as u128 * total_supply as u128 / total_assets as
    ↪  u128).try_into().unwrap();
    Ok(x)
}

pub fn convert_to_assets(&self, shares: u64, total_supply: u64) -> Result<u64>  {
    if total_supply == 0 {
        return Ok(shares);
    }

    let total_assets = self.total_assets - self.get_unvested()?; // “vested assets”:
    ↪  assumes no underflow
    let x = ((shares as u128 * total_assets as u128) / total_supply as
    ↪  u128).try_into().unwrap();
    Ok(x)
}
```

Because `get_unvested` depends on the current `vesting_period`:

```rust
pub fn get_unvested(&self) -> Result<u64> {
    let time = Clock::get()?.unix_timestamp as u32;
```

```
    let time_passed = (time - self.last_distribution_time) as u64;
    let vesting_period = self.vesting_period as u64;

    if time_passed > vesting_period {
        return Ok(0);
    }

    let amt = ((vesting_period - time_passed) * self.vesting_amount) / vesting_period;
    Ok(amt)
}
```

an admin can change `vesting_period` after rewards and withdrawals in a way that makes `get_unvested() > self.total_assets`. For example:

1. Reward R at time `t0` with some `vesting_period_old`.
2. Time advances, part of R vests; users `start_unstake`/`unstake` and withdraw the vested portion, reducing `self.total_assets` close to the remaining unvested amount under `vesting_period_old`.
3. Admin increases `vesting_period` substantially; recomputing with `vesting_period_new` makes `get_unvested()` at the current time larger than the already-reduced `self.total_assets`.

Consequenlty, `self.total_assets - self.get_unvested()?` will then panic on underflow, causing any instruction that calls `convert_to_shares`/`convert_to_assets` (e.g. `stake`, `start_unstake`) to abort with an error. This is effectively an admin-triggerable DoS of the vault's core operations.

Recommendation: It is recommended to forbid vesting changes while vesting is active.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 `Unstake` account context stack frame exceeds 4 KiB limit causing access violations

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: During compilation, Solana's BPF toolchain emits a warning that the auto-generated `try_accounts` function for `Unstake` exceeds the per-frame stack limit.

Compiler warning:

```
Function _ZN101_$LT$stake..context..Unstake$u20$as$u20$anchor_lang..Accounts$LT$stake..c
↪  ontext..UnstakeBumps$GT$GT$12try_accounts17hdf5dc2eb014773ebE Stack offset of 4184
↪  exceeded max offset of 4096 by 88 bytes, please minimize large stack variables.
↪  Estimated function frame size: 4416 bytes. Exceeding the maximum stack offset may
↪  cause undefined behavior during execution.
```

Anchor generates a `try_accounts` implementation that deserializes all of these `Account<'info, ...>` fields into locals on the stack. With this many large account structs, the frame grows beyond the BPF hard limit (4 KiB), which the toolchain explicitly flags as "may cause undefined behavior.".

In practice, this manifested during testing as intermittent

```
Access violation in stack frame ... at address ... of size ...
```

failures on `StartUnstake` / `Unstake` instructions.

Recommendation: It is recommended to minimize stack usage in `Unstake` (and similarly large contexts) by boxing account fields. For example:

```
pub struct Unstake<'info> {
    pub token_program: Program<'info, Token>,

    #[account(...)]
    pub vault_state: Box<Account<'info, VaultState>>,
```

```
    #[account(...)]
    pub staking_token: Box<Account<'info, Mint>>,

    #[account(...)]
    pub user_staking_token_account: Box<Account<'info, TokenAccount>>,

    #[account(...)]
    pub user_deposit_token_account: Box<Account<'info, TokenAccount>>,

    #[account(...)]
    pub vault_token_account: Box<Account<'info, TokenAccount>>,

    #[account(...)]
    pub blacklisted: Box<Account<'info, Blacklisted>>,

    #[account(...)]
    pub user_data: Box<Account<'info, UserPDA>>,

    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}
```

Solomon Labs: Acknowledged. Mind you when the programs were initially developed it was Anchor 0.40 AFAIK and the toolchain has changed- that is to say in practice it's not throwing, but presumably using a newer toolchain it would. So will need addressing on any update.

Cantina Managed: Acknowledged.

3.3.2 Single-step admin transfer may lead to DoS

Severity: Informational

Context: lib.rs#L497

Description: Directly setting the new admin to the supplied pubkey may lead to DoS in case the old admin makes a mistake and uses an incorrect pubkey.

Recommendation: Set a temporary value and only update the actual admin pubkey after the new admin accepts the transfer.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.3.3 Caller pubkey not emitted in `NewVaultEvent` events

Severity: Informational

Context: lib.rs#L76-L80

Description: `caller` pubkey not emitted in the event (this is against the `4.4 Events & vault identification` clause in `README.md`).

Recommendation: Add `caller` to the emitted event.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.3.4 Uncapped cooldown period

Severity: Informational

Context: lib.rs#L69, lib.rs#L137

Description: The cooldown period is set without a cap.

Recommendation: A maximum cooldown period should be checked against.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.3.5 Rewarders vector can be empty

Severity: Informational

Context: lib.rs#L247-L251

Description: The `remove_rewarder()` function does not check if at least one rewarder remains in the vector of rewarders. This allows the removal of all rewarders, leading to rewards not being distributed to participants.

Recommendation: Revert if the removal of the supplied rewarder pubkey will result in an empty vector.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.3.6 Possible overflow in value casting

Severity: Informational

Context: lib.rs#L148, lib.rs#L200

Description: Though unlikely since `rate` has to be really bigger than `DECIMALS_SCALAR`, the highlighted calculations will panic in case the resulting amount exceeds `u64::MAX`.

Recommendation: Implement better error handling instead of using `unwrap()` to avoid panics.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.3.7 Zero `redeem_rate` can disable redemptions while deposits remain enabled

Severity: Informational

Context: lib.rs#L127-L128

Description: The Vault allows the admin to set `deposit_rate` and `redeem_rate` independently via `update_asset`. In `deposit` and `redeem`, a zero rate is treated as "asset not supported".

If the admin sets `redeem_rate = 0` but leaves `deposit_rate > 0`, new deposits for that collateral remain fully allowed while all redemptions are disabled. This contradicts the intent that "rates of zero behave purely as feature toggles and never lead to stuck funds": authorized minters can continue to mint the vault token with no on-chain path to redeem for collateral (short of admin-controlled withdrawals).

Recommendation: It is recommended to treat "disabling" an asset as a symmetric operation, e.g. enforce in `update_asset` that.

```
if redeem_rate == 0 {
    require!(deposit_rate == 0, VaultError::BadRateConfig);
}
```

and introduce a clear error like `BadRateConfig`.

Solomon Labs: Acknowledged.

Cantina Managed: Acknowledged.