SECURITY ASSESSMENT

Accretion

Provided by Accretion Labs Pte Ltd. for YO Labs
April 13, 2026
A26YOL1

YO Labs

AUDITORS

Role	Name
Lead Auditor	Robert Reith (robert@accretion.xyz)
Auditor	Mahdi Rostami (mahdi@accretion.xyz)
Auditor	Niklas Brymko (niklas@accretion.xyz)

CLIENT

YO Labs (https://www.yo.xyz/) engaged Accretion to conduct a security assessment of the YO Protocol, a modular ERC4626-compliant vault system for cross-chain asset management and optimized liquidity operations on Solana

ENGAGEMENT TIMELINE

19 Mar — Project Kickoff
Initial planning and scope definition

19 Mar — Assessment Begins
Security review and testing phase

26 Mar — Review Fixes
Security recommendations are given and implemented

26 Mar — Project Completion
Report delivery and on-chain confirmation

AUDITED CODE

#	Program ID	Repository
1	yvSoLSBaLoqZ2yQttGbaYzHDXr9Bo9UdqtiRDiVaMxP	https://github.com/yoprotocol/solana
2	yvUSDCJjwvNh3jV1YwEm36Fo7oyPJBKJxHVQ6FBvbem	https://github.com/yoprotocol/solana
3	YorcLJbeA8rJcSfifPAJGcafZrvmNDwjS9sqGjb9JpW	https://github.com/yoprotocol/solana

ASSESSMENT

The security assessment of YO Labs's YO Protocol Solana revealed 15 issues with no critical or high-severity vulnerabilities. Findings included 2 medium-severity issues, 8 low-severity issues, and 5 informational items. Of the total findings, 7 issues were successfully fixed while 8 remain as won't-fix, indicating accepted risks or design choices. Medium-severity issues involved redeem request constraints and rent griefing, while low-severity findings covered validation gaps and authority management.

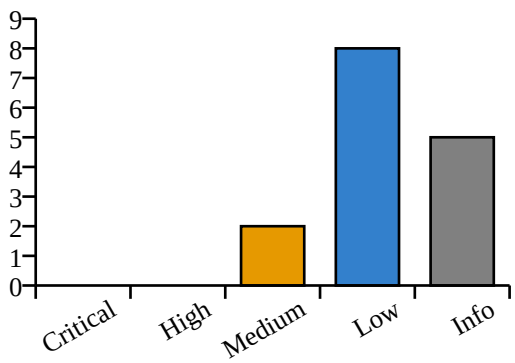
CODE ASSESSMENT

The codebase demonstrates solid architecture built on Anchor framework with appropriate use of Solana program development patterns. Security practices are generally well-implemented with proper account validations and access controls. Minor improvements needed in mint validation and authority checks.

KEY FINDINGS

No critical or high-severity vulnerabilities were identified. The two medium-severity issues involve: (1) redeem requests from PDAs becoming un-cancelable due to SystemAccount constraints, and (2) rent griefing vulnerability in fulfill/cancel redeem operations through init_if_needed patterns.

SEVERITY DISTRIBUTION



ENGAGEMENT SCOPE

The scope of this security assessment was a full review of the following items:

Item 1: YO Solana vault
Link: https://github.com/yoprotocol/solana
Commit: de2603f5dcf5c53755434d33c8aaa802b76c1a92
Program ID: yvSoLSBaLoqZ2yQttGbaYzHDXr9Bo9UdqtiRDiVaMxP
Audit Result:
- **Audited Commit:** 9f0b6a057d4a23fbdb8ba1176bb4cc0186739b2a
- **Build Hash:** 857a2e361e82e3d08e989a3d611cb2f2a93ea1876ddc32ca1b3751c0e72ff13b
- **Status:** verified

Item 2: YO USD vault
Link: https://github.com/yoprotocol/solana
Commit: de2603f5dcf5c53755434d33c8aaa802b76c1a92
Program ID: yvUSDCJjwvNh3jV1YwEm36Fo7oyPJBKJxHVQ6FBvbem
Audit Result:
- **Audited Commit:** 9f0b6a057d4a23fbdb8ba1176bb4cc0186739b2a
- **Build Hash:** 857a2e361e82e3d08e989a3d611cb2f2a93ea1876ddc32ca1b3751c0e72ff13b
- **Status:** unverified
- **Comment:** not deployed

Item 3: YO Oracle
Link: https://github.com/yoprotocol/solana
Commit: de2603f5dcf5c53755434d33c8aaa802b76c1a92
Program ID: YorcLJbeA8rJcSfifPAJGcafZrvmNDwjS9sqGjb9JpW
Audit Result:
- **Audited Commit:** 9f0b6a057d4a23fbdb8ba1176bb4cc0186739b2a
- **Build Hash:** 9932c9610e484f2dd7b162d485a54fe95b49a045db389832a36e1c437208d2bd
- **Status:** unverified
- **Comment:** on chain hash: be284c56b4d5ad7fb5f58c0040d538b07395b464136a519e1668621e2b4571cb

ISSUES SUMMARY

ID	TITLE	SEVERITY	STATUS
ACC-M1	Rent Griefing in Fulfill and Cancel Redeem via init_if_needed	MEDIUM	ACKNOWLEDGED
ACC-M2	Redeem requests from program-owned accounts (PDAs) can be made un-cancelable and un-fulfillable due to SystemAccount constraint on user	MEDIUM	FIXED
ACC-L1	No oracle price staleness check	LOW	ACKNOWLEDGED
ACC-L2	Redemption fee can change between request and fulfillment	LOW	ACKNOWLEDGED
ACC-L3	Operator controls fulfillment price within `max_pct_deviation`	LOW	ACKNOWLEDGED
ACC-L4	Passing Manager as Signer Can Allow Cross-Vault Actions	LOW	FIXED
ACC-L5	Weak manage Authority Checks Can Break Balance and Ownership Assumptions	LOW	ACKNOWLEDGED
ACC-L6	Updating vault oracle via `update_oracle` bricks deposits and redemptions due to `Program` type constraint	LOW	FIXED
ACC-L7	Insufficient Share Mint Validation in `initialize_vault` — Mint Authority Check Can Be Spoofed	LOW	ACKNOWLEDGED
ACC-L8	Missing freeze authority and supply validation on share mint during initialization	LOW	FIXED
ACC-I1	Misleading comments across the codebase	INFO	FIXED
ACC-I2	Token2022 is only supported by half of yo-libs helper functions	INFO	FIXED
ACC-I3	Program Initialization can be frontrun	INFO	ACKNOWLEDGED
ACC-I4	Partial Redemption Fills Can Drift Average Pending Price, Causing Subsequent Valid Fills to Fail	INFO	ACKNOWLEDGED
ACC-I5	Misleading comment in `unwrap_sol`: destination receives closed account lamports, not authority	INFO	FIXED

DETAILED ISSUES

ACC-M1 Rent Griefing in Fulfill and Cancel Redeem via init_if_needed

MEDIUM ACKNOWLEDGED

Description

We found that fulfill_redeem and cancel_redeem use init_if_needed for user token accounts. This creates a rent griefing vector: a user can redeem a very small amount (with negligible fee), then close their token account, and force the operator/cranker to recreate it on the next fulfill or cancel. By repeating this, an attacker can make the operator repeatedly pay rent.

Location

https://github.com/yoprotocol/solana/blob/bf1da06a80aa447a67a1d4bba16c24e6a0dbc767/programs/yo-vault/src/instructions/fulfill_redeem/fulfill_redeem.rs#L104-L112

Relevant Code

```
/// The user's deposit token account.
#[account(
  init_if_needed,
  payer = fee_payer,
  associated_token::mint = token_mint,
  associated_token::authority = user,
  associated_token::token_program = token_program
)]
pub user_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
```

Mitigation Suggestion

Do not use init_if_needed in fulfill/cancel redeem, or require the user to provide and maintain the needed token account themselves. If there is a queue for that drop the user if there is no token account.

Remediation

Acknowledged for UX reasons and due to it only affecting the async path which are fulfilled manually, allowing for active monitoring.

ACC-M2 Redeem requests from program-owned accounts (PDAs) can be made un-cancelable and un-fulfillable due to SystemAccount constraint on user

MEDIUM FIXED

Description

The `cancel_redeem`, `fulfill_redeem`, and `fulfill_redeem_native` instructions all declare the `user` account as `SystemAccount<'info>`, which enforces that the account must be owned by the System Program. However, the `request_redeem` instruction declares `user` as merely `Signer<'info>`, which allows any account — including PDAs owned by other programs — to submit a redeem request.

This creates an asymmetry: a program-owned account (PDA) can successfully call `request_redeem` via CPI (where the owning program provides the signature), escrowing shares into the `minter_share_vault` and recording a pending redeem. However, when the operator subsequently tries to cancel or fulfill that redeem request, the `SystemAccount` deserialization check on the `user` field will fail because the PDA's owner is not the System Program.

Attack scenario:

1. A user deploys a custom program that owns a PDA.

2. The custom program calls `request_redeem` via CPI, passing the PDA as the `user` signer. This succeeds because `request_redeem` only requires `Signer`.

3. Shares are escrowed and a `UserPendingRedeem` PDA is created, keyed by the PDA's public key.

4. The operator attempts to call `cancel_redeem` for this user — it fails because the `user` account (the PDA) is not owned by the System Program, violating the `SystemAccount` constraint.

5. Similarly, `fulfill_redeem` and `fulfill_redeem_native` also fail for the same reason.

Alternatively, a malicious user can create a redeem request with a regular SystemAccount, and then reassign ownership of that account to another program, effectively "orphaning" the redeem request and making it un-cancelable and un-fulfillable. They can reverse the ownership change afterward to make the request functional again, creating an at-will denial of service vector against their own redeem requests.

Impact:

• **Denial of Service on Vault Operations:** The pending redeem amount is added to `vault_data.total_pending_redeems`, which permanently inflates this counter since the redeem can never be cancelled or fulfilled. This reduces the `available` balance in future `request_redeem` calls (line 183 of `request_redeem.rs`: `vault_token_balance.saturating_sub(vault_data.total_pending_redeems)`), artificially forcing subsequent legitimate redeems into the escrow path instead of immediate fulfillment.

• **Permanent Share Lock:** The escrowed shares in the `minter_share_vault` become permanently locked — they can never be returned to the user (cancel) or burned (fulfill).

• **Griefing:** A malicious user can repeatedly create such un-resolvable redeem requests to progressively degrade vault functionality by inflating `total_pending_redeems`.

Location

https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/programs/yo-vault/src/instructions/cancel_redeem.rs#L0

Also affected:

• https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/programs/yo-vault/src/instructions/fulfill_redeem/fulfill_redeem.rs#L28

• https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/programs/yo-vault/src/instructions/fulfill_redeem/fulfill_redeem_native.rs#L31

Relevant Code

`cancel_redeem.rs` — user declared as `SystemAccount`:

```
/// The user that originally requested the redeem and that will receive the refunded shares.

#[account(mut)]

pub user: SystemAccount<'info>,
```

`fulfill_redeem.rs` — same constraint:

```
/// The user receiving the redeemed lamports.

#[account(mut)]

pub user: SystemAccount<'info>,
```

`request_redeem.rs` — user declared as mere `Signer`, allowing PDAs:

```
/// The user requesting the redemption and receiving SOL when redeem is fulfilled.

#[account(mut)]

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

Mitigation Suggestion

Change `user` from `SystemAccount<'info>` to `UncheckedAccount<'info>` (with `/// CHECK:` safety doc) in `cancel_redeem`, `fulfill_redeem`, and `fulfill_redeem_native`. The `user` account in these instructions is not a signer and is only used as an authority reference for the associated token account derivation and as a transfer destination. The `pending_redeem_state` PDA seed derivation already validates the relationship between the user key and the pending redeem. Care should be taken to verify that the `init_if_needed` ATA creation and token transfers still function correctly for program-owned accounts.

Remediation

Fixed in https://github.com/yoprotocol/solana/pull/71.

ACC-L1 No oracle price staleness check

LOW ACKNOWLEDGED

Description

Both `deposit` and `request_redeem` read `oracle_vault_data.last_price` without checking `oracle_vault_data.last_slot_updated`. The field `last_slot_updated` is written by the oracle (`update_vault_price.rs:71`, `initialize_vault.rs:85`) but is never read by any consumer in the vault program.

The admin `pause` mechanism is the only mitigation, but it is reactive/manual with an inherent detection gap between oracle failure and pause activation.

Location

https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/instructions/deposit.rs#L147-L151 https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/instructions/request_redeem.rs#L173-L177

Relevant Code

```
/// programs/yo-vault/src/instructions/deposit.rs L147-L151
let (shares, fee) = vault_lib::preview_deposit(
    assets,
    ctx.accounts.oracle_vault_data.last_price,  // no staleness check
    vault_data.pct_deposit_fee,
)?;



/// programs/yo-vault/src/instructions/request_redeem.rs L173-L177
let (assets, fee) = vault_lib::preview_redeem(
    shares,
    ctx.accounts.oracle_vault_data.last_price,  // no staleness check
    vault_data.pct_redemption_fee,
)?;
```

Mitigation Suggestion

Add a maximum staleness check before using the oracle price. Both `deposit` and `request_redeem` should verify that the oracle was updated recently:

```
let clock = Clock::get()?;
require!(
    clock.slot.saturating_sub(ctx.accounts.oracle_vault_data.last_slot_updated) <= MAX_PRICE_AGE_SLOTS,
    YoError::StalePrice
);
```

`MAX_PRICE_AGE_SLOTS` should be a configurable parameter on `VaultData` so the admin can tune it per deployment.

Remediation

Acknowledged.

Yo Labs: This is also something we considered but we decided it was a better user experience to continue to allow deposits and withdraws at a slightly worse rate rather than throwing the transaction for the user. Especially as the vault

price is relatively stable and ticks up quite slowly over time. Users should always be able to withdraw their assets.

ACC-L2 Redemption fee can change between request and fulfillment

LOW ACKNOWLEDGED

Description

`UserPendingRedeem` stores only `pending_assets` and `pending_shares` - there is no field for a snapshotted fee percentage. In the escrow path of `request_redeem`, the fee is emitted as `0` in the event and never stored. At fulfillment time, the fee is recomputed from the current `vault_data.pct_redemption_fee`.

The admin can change `pct_redemption_fee` at any time via `update_redemption_fee` with no guard against pending requests.

Location

https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/instructions/fulfill_redeem/fulfill_redeem.rs#L153

Relevant Code

```
/// programs/yo-vault/src/instructions/fulfill_redeem/fulfill_redeem.rs L153
let fee = vault_lib::fee_on_raw(assets, ctx.accounts.vault_data.pct_redemption_fee, true)?;
```

Mitigation Suggestion

Snapshot the fee at request time by adding a `pct_redemption_fee` field to `UserPendingRedeem`:

Remediation

Acknowledged as a deliberate design choice, charging fees at redemption time.

ACC-L3 Operator controls fulfillment price within `max_pct_deviation`

LOW ACKNOWLEDGED

Description

The operator chooses the `shares` and `assets` parameters for fulfillment. The only constraint is that the implied price is within `max_pct_deviation` (up to 10%) of the average price of pending requests. The `percentage_change` function uses `abs_diff` (symmetric), allowing deviation in either direction.

The `LeftoverUnfulfilled` check only constrains the terminal state (both zero or both nonzero). It does NOT prevent partial fulfillment at a bad price. Validated attack path:

1. Partial fulfill: `fulfill_redeem(shares=999, assets=901)` for a user with `pending_shares=1000, pending_assets=1000` (~9.9% underpayment) 2. Cancel remainder: `cancel_redeem()` returns the 1 remaining share to user 3. Net loss to user: ~99 assets (~9.9%)

Location

https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/instructions/fulfill_redeem/common.rs#L29-L37

Relevant Code

```
/// programs/yo-vault/src/instructions/fulfill_redeem/common.rs L29-L37
let avg_pending_price =
    vault_lib::share_price(pending.pending_assets, pending.pending_shares, false)?;
let redemption_price = vault_lib::share_price(assets, shares, false)?;
let pct_deviation = math_lib::percentage_change(avg_pending_price, redemption_price, true)?;
require!(
    pct_deviation <= vault_data.max_pct_deviation as u128,
    YoError::PriceChangeTooLarge
);
```

Mitigation Suggestion

Instead of passing both `shares` and `assets`, derive `assets` from the locked-in pending ratio. The price was already determined at request time via the oracle, so fulfillment should honor that ratio:

```
// operator only passes `shares` to fulfill
// assets derived from the locked-in ratio
let assets = mul_div(shares, pending.pending_assets, pending.pending_shares, false)?;
```

This makes every partial fulfillment proportional by construction. The partial-fulfill + cancel attack no longer works since the operator cannot choose a worse price. The `max_pct_deviation` parameter becomes unnecessary for this path.

Remediation

Acknowledged to maintain consistency with EVM contracts and the existing backend infrastructure.

ACC-L4 Passing Manager as Signer Can Allow Cross-Vault Actions

LOW FIXED

Description

We found that manage.rs passes manager as a signer to the target program. While Solana does not have classic reentrancy, multiple vaults can share the same manager. Because of this, if the target program receives the manager as a signer, it may use that signer privilege to manage other vaults controlled by the same manager, not just the intended vault.

Location

https://github.com/yoprotocol/solana/blob/bf1da06a80aa447a67a1d4bba16c24e6a0dbc767/programs/yo-vault/src/instructions/manage/common.rs#L16-L28

Relevant Code

```
/// via `remaining_accounts`.  The authority and manager accounts are marked as signer.
pub fn get_accounts(
  remaining_accounts: &[AccountInfo],
  authority: &Pubkey,
  manager: &Pubkey,
) -> Result<Vec<AccountMeta>> {
  let accounts: Vec<AccountMeta> = remaining_accounts
    .iter()
    .map(|acc| {
      let is_signer = acc.key == authority || acc.key == manager;
      AccountMeta {
        pubkey: *acc.key,
        is_signer,
```

Mitigation Suggestion

Do not pass manager as a signer to the target program unless it is strictly required.

Remediation

Fixed in https://github.com/yoprotocol/solana/pull/72 and https://github.com/yoprotocol/solana/pull/76 by using fee payer instead of manager.

ACC-L5 Weak manage Authority Checks Can Break Balance and Ownership Assumptions

LOW ACKNOWLEDGED

Description

We found that in manage.rs we pass authority as writable, and the current authority checks in manage.rs are not strong enough and can lead to multiple issues:

1. During unwrap SOL flows, authority can temporarily hold SOL, but manage.rs does not check that the SOL balance is unchanged before and after the call. 2. In many instructions, authority is used as a system account, but there is no check that its owner remains unchanged. If the target program changes the account owner, later protocol logic may stop working. 3. Authority is also used as the authority of token accounts, but there is no check that the token account authority remains correct. 4. Authority can have multiple token accounts and we check the balance on input and output token accounts; there is no check that the actual token accounts used in CPI are the input and output token accounts.

Because of this, external calls can violate important assumptions about balance, ownership, and token authority.

Location

https://github.com/yoprotocol/solana/blob/bf1da06a80aa447a67a1d4bba16c24e6a0dbc767/programs/yo-vault/src/instructions/manage/manage.rs#L29-L35

Relevant Code

```
/// The authority PDA for signing the call.
  #[account(
    mut,
    seeds=[YO_VAULT_AUTH_SEED.as_bytes()],
    bump
  )]
  pub authority: SystemAccount<'info>,
```

Mitigation Suggestion

Pass authority as non-mutable, so in this way, we verify the Sol balance and the owner of the PDA. For token accounts, check that only the balances change.

Remediation

Acknowledged by the team, manager is a trusted role, and authority mutability is needed to transfer native SOL from the authority account.

ACC-L6 Updating vault oracle via `update_oracle` bricks deposits and redemptions due to `Program` type constraint

LOW FIXED

Description

The `update_oracle` function in `update_config.rs` allows the admin to change the oracle program address stored in `vault_data.oracle` to an arbitrary `Pubkey`. However, the `deposit` and `request_redeem` instructions declare the oracle account using Anchor's `Program<'info, YoOracle>` type, which performs a **compile-time program ID check** against the hardcoded `declare_id!("YorcLJbeA8rJcSfifPAJGcafZrvmNDwjS9sqGjb9JpW")` from the `yo_oracle` crate.

This creates an irreconcilable conflict:

1. `Program<'info, YoOracle>` **deserialization** — Anchor automatically validates that the provided account's key matches the program ID declared by `YoOracle` (i.e., `YorcLJbeA8rJcSfifPAJGcafZrvmNDwjS9sqGjb9JpW`). This check is baked into the type system and cannot be overridden at runtime.

2. `constraint = yo_oracle.key() == vault_data.oracle` — This runtime constraint checks that the oracle account matches whatever address is stored in `vault_data.oracle`.

If the admin calls `update_oracle` with a new oracle address that differs from `YorcLJbeA8rJcSfifPAJGcafZrvmNDwjS9sqGjb9JpW`, the two checks become mutually exclusive:

- Passing the original oracle program satisfies `Program<YoOracle>` but **fails** the `vault_data.oracle` constraint.
- Passing the new oracle address satisfies the `vault_data.oracle` constraint but **fails** the `Program<YoOracle>` deserialization.

No valid account can satisfy both checks simultaneously, meaning **deposits and redemption requests are permanently bricked** after an oracle update to a different program ID. The vault's core functionality (depositing and redeeming) becomes completely inoperable with no on-chain recovery path other than updating the oracle back to the original address.

This is particularly dangerous because:

- The `update_oracle` function has no guard preventing it from accepting an incompatible address — it only checks for `Pubkey::default()` and unchanged values.
- An admin mistake or a legitimate need to migrate to a new oracle program would freeze user funds in the vault until the oracle is switched back or the program is upgraded.
- The function appears fully functional and would emit a successful `EventUpdateOracle` event, giving no indication that the vault is now bricked.

Location

https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/programs/yo-vault/src/instructions/update_config.rs#L0

Relevant Code

`update_config.rs` — the oracle update function that allows setting an arbitrary address:

```
pub fn update_oracle(ctx: Context<UpdateConfig>, new_oracle: Pubkey) -> Result<()> {

  let vault_data: &mut Account<'_, VaultData> = &mut ctx.accounts.vault_data;

  let old_oracle = vault_data.oracle;
```

```
  require!(new_oracle != Pubkey::default(), YoError::NullAddress);
  require!(new_oracle != old_oracle, YoError::Unchanged);

  vault_data.oracle = new_oracle;
  // ...
}
```

`deposit.rs` (lines 125-127) and `request_redeem.rs` (lines 147-149) — the conflicting type + constraint:

```
/// YoOracle program
#[account(constraint = yo_oracle.key() == vault_data.oracle)]
pub yo_oracle: Program<'info, YoOracle>,
```

The `YoOracle` program type is pinned to:

```
declare_id!("YorcLJbeA8rJcSfifPAJGcafZrvmNDwjS9sqGjb9JpW");
```

Mitigation Suggestion

There are several possible approaches depending on the intended design:

1. **Remove `update_oracle` entirely**: If the oracle program is meant to be immutable (as the `Program<YoOracle>` type implies), remove the `update_oracle` function to eliminate the misleading admin capability. The runtime constraint `yo_oracle.key() == vault_data.oracle` is also redundant in this case since `Program<YoOracle>` already enforces the correct program ID.

2. **Use `UncheckedAccount` instead of `Program<YoOracle>`**: If the oracle truly needs to be updatable, replace `Program<'info, YoOracle>` with an `UncheckedAccount` (or `AccountInfo`) and rely solely on the `vault_data.oracle` constraint for validation.

3. **Validate compatibility in `update_oracle`**: Add a check in `update_oracle` that ensures the new oracle address is actually an executable program account, and consider restricting updates to only be possible when the vault is paused, giving users time to withdraw before any migration.

Option 1 is the simplest and safest if oracle upgradability is not a true requirement. If it is needed, option 2 combined with option 3 provides the most flexibility while maintaining safety.

Remediation

Fixed in https://github.com/yoprotocol/solana/pull/69 by removing the instruction.

ACC-L7 Insufficient Share Mint Validation in `initialize_vault` — Mint Authority Check Can Be Spoofed

LOW ACKNOWLEDGED

Description

In the `yo-oracle` program's `initialize_vault` instruction, the `vault_share_mint` account is validated by checking that its `mint_authority` matches the expected vault minter PDA derived from the provided `vault_program`. However, this check is insufficient because **anyone can create a new SPL token mint and set its mint authority to any arbitrary address**, including the vault minter PDA.

The current validation logic (lines 70–77) is:

```
let (vault_minter, _) = Pubkey::find_program_address(

  &[YO_VAULT_MINTER_SEED.as_bytes()],

  &ctx.accounts.vault_program.key(),

);

require!(

  vault_minter == ctx.accounts.vault_share_mint.mint_authority.unwrap(),

  YoError::IncorrectShareMint

);
```

This only proves that the mint's authority *happens to be* the vault minter PDA — it does **not** prove that the mint is the *actual* share mint registered with the vault program. An attacker (or a misconfiguration by the admin) could pass in a freshly created mint whose authority was set to the vault minter PDA, and it would pass this check despite not being the legitimate share mint stored in the vault program's `VaultData` account.

Additionally, the `vault_program` account itself is an `UncheckedAccount` (line 39) with no validation that it is actually an executable program or a known vault program ID. A malicious `vault_program` key could be crafted such that the derived minter PDA matches the mint authority of an attacker-controlled mint.

The correct approach is to deserialize the vault program's `VaultData` PDA account (derived with seeds `["yo_vault_VaultData"]` under the provided `vault_program`) and read the `share_mint` field directly from it. This provides a cryptographic guarantee that the share mint being registered in the oracle is the same one that the vault program considers canonical.

Security impact: If a malicious or incorrect share mint is registered in the oracle, subsequent price updates via `update_vault_price` would be applied against the wrong mint. Since the oracle's price data feeds into the vault's deposit and redemption calculations, this could lead to incorrect share pricing — potentially enabling fund extraction at manipulated exchange rates. The severity is partially mitigated by the fact that `initialize_vault` is gated to the oracle admin, but the defense-in-depth principle warrants proper validation regardless of caller privilege.

Location

https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/programs/yo-oracle/src/instructions/initialize_vault.rs#L0

Relevant Code

```
/// Address of the vault share token.  Used as key to derive the vault data account PDA.
```

```
pub vault_share_mint: Box<InterfaceAccount<'info, Mint>>,

// ...

/// CHECK: The vault program.
pub vault_program: UncheckedAccount<'info>,

// ...

// Check vault program is correct
// We expect the vault minter to be the mint authority of the share token
let (vault_minter, _) = Pubkey::find_program_address(
  &[YO_VAULT_MINTER_SEED.as_bytes()],
  &ctx.accounts.vault_program.key(),
);
require!(
  vault_minter == ctx.accounts.vault_share_mint.mint_authority.unwrap(),
  YoError::IncorrectShareMint
);
```

Mitigation Suggestion

1. **Derive and deserialize the vault's** `VaultData` **PDA** from the provided `vault_program`, then read the `share_mint` field from it. Validate that the passed `vault_share_mint` matches this on-chain canonical value:

```
// Add the vault's VaultData account to the accounts struct:
/// The vault data account from the vault program, used to read the canonical share mint.
#[account(
  seeds = [b"yo_vault_VaultData"],
  bump,
  seeds::program = vault_program.key(),
)]
pub vault_vault_data: Account<'info, VaultData>,
```

Then in the instruction body, replace the mint authority check with:

```
require!(
  ctx.accounts.vault_share_mint.key() == ctx.accounts.vault_vault_data.share_mint,
  YoError::IncorrectShareMint
);
```

2. **Validate the** `vault_program` **account** is an executable program, or better yet, constrain it to a known set of vault program IDs to prevent arbitrary program keys from being supplied.

3. The existing mint-authority check can optionally be retained as an additional layer of validation, but it should not be the sole check.

Remediation

Client has acknowledged the issue, and stated that no change will be made as it would break the account interface.

ACC-L8 Missing freeze authority and supply validation on share mint during initialization

LOW FIXED

Description

During vault initialization of a vault in `initialize.rs`, the `share_mint` account is validated only for its `mint_authority` (line 53), ensuring it has been delegated to the vault's minter PDA. However, two critical properties of the share mint are not validated:

1. **Freeze Authority is not checked to be `None`:** If the share mint was created with a freeze authority set, the holder of that authority could freeze any share token account at will. This would prevent users from transferring or redeeming their share tokens, effectively locking their deposited funds in the vault. Since the share mint is created externally before program deployment (as noted in the code comment: *"created before program deployment to support a premined vanity address"*), there is no programmatic guarantee that the freeze authority was set to `None` at mint creation time.

2. **Supply is not checked to be zero:** The share mint is created externally before initialization. If the mint authority holder minted tokens before delegating authority to the minter PDA and calling `initialize`, there would be outstanding share tokens not backed by any deposited assets. These pre-minted shares would be redeemable against future deposits, allowing the pre-minter to extract value from legitimate depositors. The share-to-asset exchange rate calculations in the deposit and redeem flows rely on `share_mint.supply` being accurate relative to the vault's total assets — a non-zero initial supply would corrupt this invariant from the start.

Both of these issues arise because the share mint is created out-of-band (externally) rather than by the program itself, yet the initialization instruction does not sufficiently validate the mint's state before accepting it.

Severity: Low — in practice, the deployer controls the share mint creation and initialization sequence, and the `initialize` endpoint is unguarded (first-caller-wins). However, defense-in-depth dictates that these invariants should be enforced on-chain rather than relying on correct off-chain operational procedures.

Location

https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/programs/yo-vault/src/instructions/initialize.rs#L0

Relevant Code

```
/// The share mint, created before program deployment to support a premined vanity address.
/// It must delegate its mint authority to the vault minter PDA before calling `initialize`.
/// This is checked inside the Anchor constraint.
#[account(
  constraint = share_mint.mint_authority.unwrap() == minter.key(),
  mint::token_program = token_program
)]
pub share_mint: Box<InterfaceAccount<'info, Mint>>,
```

Mitigation Suggestion

Add Anchor constraints to validate that the share mint's `freeze_authority` is `None` and that its `supply` is `0` at initialization time:

```
  #[account(
    constraint = share_mint.mint_authority.unwrap() == minter.key(),
    constraint = share_mint.freeze_authority.is_none() @ YoError::InvalidMintFreezeAuthority,
    constraint = share_mint.supply == 0 @ YoError::InvalidMintSupply,
    mint::token_program = token_program
  )]
  pub share_mint: Box<InterfaceAccount<'info, Mint>>,
```

Add the corresponding error variants to `YoError`:

```
#[error_code]
pub enum YoError {
    // ... existing variants ...
    #[msg("Share mint freeze authority must be None")]
    InvalidMintFreezeAuthority,
    #[msg("Share mint supply must be zero at initialization")]
    InvalidMintSupply,
}
```

This ensures that even if the share mint was misconfigured during its external creation, the vault program will reject it during initialization rather than silently accepting a potentially dangerous state.

Remediation

Fixed in https://github.com/yoprotocol/solana/pull/68.

ACC-I1 Misleading comments across the codebase

INFO FIXED

Description

Multiple documentation comments across both `yo-vault` and `yo-oracle` misidentify which privileged role is required for an instruction. These do not affect runtime behavior (the on-chain constraints are correct), but create confusion for integrators, auditors, and future maintainers.

yo-vault/`lib.rs`: `manage*` instructions labeled "Operator only", actually require Manager

`lib.rs` lines 94, 114, 127, 140 all say `/// Operator only` for the four `manage` variants. The actual account constraints check `vault_data.manager`:

```
/// lib.rs L94 - comment says:
/// Operator only
/// Manage vault to call external programs.

/// manage.rs L22 - actual constraint:
#[account(mut, constraint = manager.key() == vault_data.manager)]
pub manager: Signer<'info>,
```

Affected instructions: `manage`, `manage_unchecked_output`, `manage_unchecked_input`, `manage_unchecked`.

yo-vault/`vault_data.rs`: `manager` field documented as "The operator"

```
/// vault_data.rs L22
/// The operator with privileged access to manage the vault (invest, divest, bridge etc).
pub manager: Pubkey,
```

Should say "The manager".

yo-oracle Four admin-only instructions documented as "Program updater"

The following instructions all check `oracle_config.admin` but their doc comments say "Program updater":

- `initialize_vault.rs:14` "Program updater with privilege to initialize vaults"
- `update_config.rs:12` - "Program updater with privilege to update oracle configs"
- `update_vault_config.rs:14` - "Program updater with privilege to update vault configs"
- `set_updater.rs:11` - "Program updater with privilege to set the updater"

Location

- https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/lib.rs#L94-L150
- https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/state/vault_data.rs#L22
- https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-oracle/src/instructions/initialize_vault.rs#L14
- https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-oracle/src/instructions/update_config.rs#L12
- https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-oracle/src/instructions/update_vault_config.rs#L14
- https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-oracle/src/instructions/set_updater.rs#L11
- https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/events.rs#L29

Relevant Code

Mitigation Suggestion

Update all comments to match the actual on-chain access control constraints.

Remediation

Fixed in https://github.com/yoprotocol/solana/pull/75.

ACC-I2 Token2022 is only supported by half of yo-libs helper functions

INFO FIXED

Description

We found that the `mint` and `burn` `yo-lib` library helper uses the `token::` functions instead of `token_interface::` like transfer and wSOL un/wrapping.

Location

https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/crates/yo-lib/src/libraries/token.rs#L79-L116

Relevant Code

```
/// token.rs L79-L116
  token::mint_to(
    CpiContext::new_with_signer(
      token_program.to_account_info(),
      token::MintTo {
        mint,
        to,
        authority,
      },
      signer_seeds,
    ),
    amount,
  )
}

/// Burn tokens from an account.
pub fn burn<'a>(
  mint: AccountInfo<'a>,
  from: AccountInfo<'a>,
  authority: AccountInfo<'a>,
  amount: u64,
  token_program: AccountInfo<'a>,
  signer_seeds: &[&[&[u8]]],
) -> Result<()> {
  if amount == 0 {
    return Err(YoError::ZeroAmount.into());
  }
  token::burn(
    CpiContext::new_with_signer(
      token_program.to_account_info(),
      token::Burn {
        mint,
        from,
        authority,
      },
      signer_seeds,
    ),
    amount,
  )
```

Mitigation Suggestion

These instructions also can be triggered with the `token_interface::` namespace, similar to other helper in `yo-lib`

Remediation

Fixed in https://github.com/yoprotocol/solana/pull/73.

ACC-I3 Program Initialization can be frontrun

INFO ACKNOWLEDGED

Description

We found that both programs' `initialize` instructions have no access control. The code comments explicitly acknowledge this as a known design choice. Anchor's `init` constraint prevents re-initialization but not first-mover capture.

This is preventable by either a hard-coded initializer key or by allowing only the upgrade authority to call this instruction.

Location

https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-vault/src/instructions/initialize.rs#L18-L31 https://github.com/yoprotocol/solana/blob/dfd046ecba0f1cd4ea4689402f75977646b1b61e/programs/yo-oracle/src/instructions/initialize.rs#L12-L25

Relevant Code

```
/// programs/yo-vault/src/instructions/initialize.rs L18-L31
/// Program deployer that sets initial admin and pays for account initialization.
/// Note: Endpoint is not guarded so must be called by deployer immediately after deployment.
#[account(mut)]
pub caller: Signer<'info>,
```

Mitigation Suggestion

Require the caller to be the program's upgrade authority or a build-time hardcoded pubkey:

Option A: Verify caller is the program's upgrade authority at runtime

```
#[account(address = crate::ID)]
#[account(constraint = program.programdata_address() == Ok(Some(program_data.key())))]
pub program: Program<'info, crate::program::YoVault>,

#[account(constraint = program_data.upgrade_authority_address == Some(caller.key()))]
pub program_data: Account<'info, ProgramData>,
```

Option B: Hardcode the deployer pubkey at build time

```
#[account(mut, address = DEPLOYER_PUBKEY)]
pub caller: Signer<'info>,
```

Remediation

Acknowledged.

ACC-I4 Partial Redemption Fills Can Drift Average Pending Price, Causing Subsequent Valid Fills to Fail

INFO ACKNOWLEDGED

Description

The deviation check in `fulfill_redeem_common` compares each fill's price against the **current remaining** average pending price (`pending.pending_assets / pending.pending_shares`), rather than the **original** average price at the time the redemption was requested. This means that after a partial fill at a price below the average, the remaining average price shifts upward, and subsequent fills at the same (originally-valid) price may exceed the `max_pct_deviation` threshold and revert.

Concrete Example (5% max deviation):

1. User requests redemption: **10 shares, 1000 assets** → avg price = **100.0** assets/share

2. Operator fills **1 share at 95 assets** (5.0% deviation from 100.0 → passes)

3. Remaining state: **9 shares, 905 assets** → new avg price = **905/9** ≈ **100.556** assets/share

4. Operator attempts to fill another **1 share at 95 assets** → deviation from 100.556 is **~5.53%** → **FAILS**

Even though the fill price of 95 assets/share was within the allowed deviation of the *original* redemption price (100 assets/share), the second fill fails because the remaining average has drifted upward. Each below-average partial fill shifts the remaining average further away, creating a compounding effect that progressively tightens the effective deviation window.

This has several consequences:

• **Legitimate fills can fail**: An operator fulfilling at a consistent, valid price may be blocked after the first partial fill.

• **Order-dependent behavior**: The success of fills depends on the sequence in which partial fills are processed. Filling the "cheap" shares first poisons the remaining average for subsequent fills.

• **Potential griefing / operational disruption**: If an operator is constrained to fill at market prices that are slightly below the original average, they may be unable to complete the redemption at all, forcing a cancellation and re-request cycle.

• **Asymmetric effect**: Below-average fills make the remaining average *higher* (harder to fill below), while above-average fills make the remaining average *lower* (harder to fill above). This creates a ratchet effect in either direction.

Location

https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/programs/yo-vault/src/instructions/fulfill_redeem/common.rs#L0

Relevant Code

```
// fulfill_redeem/common.rs lines 30-37

let avg_pending_price =

  vault_lib::share_price(pending.pending_assets, pending.pending_shares, false)?;

let redemption_price = vault_lib::share_price(assets, shares, false)?;

let pct_deviation = math_lib::percentage_change(avg_pending_price, redemption_price, true)?;

require!(
```

```
  pct_deviation <= vault_data.max_pct_deviation as u128,
  YoError::PriceChangeTooLarge
);

// State update subtracts filled amounts, shifting the remaining average
pending.pending_shares = pending
  .pending_shares
  .checked_sub(shares)
  .ok_or(YoError::SubOverflow)?;
pending.pending_assets = pending
  .pending_assets
  .checked_sub(assets)
  .ok_or(YoError::SubOverflow)?;
```

Mitigation Suggestion

Consider one of the following approaches:

1. **Store and check against the original average price**: Record the original average price (or the original total shares/assets at the time of the redemption request) in the `UserPendingRedeem` account and compare each fill against that fixed reference price, rather than the drifting remaining average. This ensures that all fills within the original deviation window are accepted regardless of fill order.

2. **Check deviation per-fill against the initial request price**: Add an `original_share_price` field to `UserPendingRedeem` that is set once when the redemption is first requested (or computed as a weighted average when additional requests are merged). Use this as the baseline for all deviation checks.

3. **Accumulate and check the blended fill price**: Instead of checking each fill individually against the remaining average, track the total assets and shares *already fulfilled* and check whether the cumulative blended fill price (total assets filled / total shares filled) stays within deviation of the original request price.

Option 2 is the most straightforward: store `original_price` once, compare each partial fill against it, and the check becomes order-independent and stable across arbitrary partial fill sequences.

Remediation

Achnowledged. Yo Labs: The share price ticks up very slowly over time (single digit % per year) and we aim to fill requests within hours so this is not a realistic issue in practice.

ACC-I5 Misleading comment in `unwrap_sol`: destination receives closed account lamports, not authority

INFO FIXED

Description

In the `unwrap_sol` helper function in `yo-lib`, the comment on line 185 states:

```
// can be paid by authority as it received the rent from account closure
```

This comment is misleading. When `token_interface::close_account` is called (line 172-180), the lamports from the closed token account (including rent) are sent to the `destination` parameter, **not** the `authority` parameter. The `authority` parameter in `CloseAccount` is the account that has the authority to close the token account, while `destination` is the account that receives the reclaimed lamports.

However, in the only current call site (`programs/yo-vault/src/instructions/unwrap_sol.rs:73-74`), the `authority` and `destination` arguments are both set to `ctx.accounts.authority.to_account_info()` — the same account. This means the code functions correctly in practice because the vault authority PDA receives the lamports (as `destination`) and then uses those lamports to pay for the ATA re-creation (as `payer`/`authority`).

Severity: Informational / Low

While the current behavior is correct because both parameters resolve to the same account, the misleading comment creates a risk for future maintainers:

• If a future caller of `unwrap_sol` passes **different** accounts for `destination` and `authority`, the `authority` would **not** have received the rent lamports from the account closure, and the `associated_token::create` call on line 186 could fail due to insufficient lamports in the `authority` account (or drain lamports that don't belong to it).

• The library function's API does not enforce that `destination == authority`, so this implicit assumption is not documented or validated.

Location

https://github.com/yoprotocol/solana/blob/de2603f5dcf5c53755434d33c8aaa802b76c1a92/crates/yo-lib/src/libraries/token.rs#L185

Relevant Code

```
// unwrap all wSOL

token_interface::close_account(CpiContext::new_with_signer(

    token_program.to_account_info(),

    token_interface::CloseAccount {

        account: account.to_account_info(),

        destination: destination.clone(),   // <-- this account receives lamports

        authority: authority.clone(),        // <-- this is the close authority, NOT the recipient

    },

    signer_seeds,

))?;


// if remaining wSOL balance exceeds requested amount, re-wrap the difference
```

```
if remaining_wsol_balance > 0 {
    // re-create the ATA (was closed above)
    // can be paid by authority as it received the rent from account closure  <-- MISLEADING
    associated_token::create(CpiContext::new_with_signer(
        associated_token_program.to_account_info(),
        associated_token::Create {
            payer: authority.clone(),   // <-- assumes authority has the lamports, but destination received t
hem
            ...
        },
        signer_seeds,
    ))?;
```

Mitigation Suggestion

1. **Fix the comment** to accurately reflect that it is `destination` that receives the lamports from account closure, and note the implicit requirement that `destination` and `authority` must be the same account (or that `authority` must have sufficient lamports independently):

```rust
// re-create the ATA (was closed above)
// authority must have sufficient lamports to pay rent (in practice, authority == destination,
// so it received the rent back from the account closure above)
```

2. **Consider adding a validation** at the top of `unwrap_sol` to enforce this assumption if it is indeed required:

```rust
require!(destination.key() == authority.key(), YoError::InvalidAccount);
```

Alternatively, use `destination` as the payer for the ATA re-creation instead of `authority`, which would be correct regardless of whether the two accounts are the same.

3. **Apply the same logic to the `else` branch** (lines 209-222): the rent refund is transferred from `authority` to `fee_payer`, but `destination` (not `authority`) received the lamports from closure. The same implicit `destination == authority` assumption applies there.

Remediation

Fixed in https://github.com/yoprotocol/solana/pull/70.

APPENDIX

Vulnerability Classification

We rate our issues according to the following scale. Informational issues are reported informally to the developers and are not included in this report.

Severity	Description
Critical	Vulnerabilities that can be easily exploited and result in loss of user funds, or directly violate the protocol's integrity. Immediate action is required.
High	Vulnerabilities that can lead to loss of user funds under non-trivial preconditions, loss of fees, or permanent denial of service that requires a program upgrade. These issues require attention and should be resolved in the short term.
Medium	Vulnerabilities that may be more difficult to exploit but could still lead to some compromise of the system's functionality. For example, partial denial of service attacks, or such attacks that do not require a program upgrade to resolve, but may require manual intervention. These issues should be addressed as part of the normal development cycle.
Low	Vulnerabilities that have a minimal impact on the system's operations and can be fixed over time. These issues may include inconsistencies in state, or require such high capital investments that they are not exploitable profitably.
Informational	Findings that do not pose an immediate risk but could affect the system's efficiency, maintainability, or best practices.

Audit Methodology

Accretion is a boutique security auditor specializing in Solana's ecosystem. We employ a customized approach for each client, strategically allocating our resources to maximize code review effectiveness. Our auditors dedicate substantial time to developing a comprehensive understanding of each program under review, examining design decisions, expected and edge-case behaviors, invariants, optimizations, and data structures, while meticulously verifying mathematical correctness—all within the context of the developers' intentions.

Our audit scope extends beyond on-chain components to include associated infrastructure, such as user interfaces and supporting systems. Every audit encompasses both a holistic protocol design review and detailed line-by-line code analysis.

During our assessment, we focus on identifying:
• Solana-specific vulnerabilities
• Access control issues
• Arithmetic errors and precision loss
• Race conditions and MEV opportunities
• Logic errors and edge cases
• Performance optimization opportunities
• Invariant violations
• Account confusion vulnerabilities
• Authority check omissions
• Token22 implementation risks and SPL-related pitfalls
• Deviations from best practices

Our approach transcends conventional vulnerability classifications. We continuously conduct ecosystem-wide security research to identify and mitigate emerging threat vectors, ensuring our audits remain at the forefront of Solana security practices.