// Security Assessment

03.02.2026 - 03.06.2026

Glow Vault: Transferrable Tokens & Epoch Withdrawal

Blueprint Finance

HALBORN

Glow Vault: Transferrable Tokens & Epoch Withdrawal - Blueprint Finance

Prepared by: HALBORN

Last Updated 04/20/2026

Date of Engagement: March 2nd, 2026 - March 6th, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	0	1	4	5

TABLE OF CONTENTS

8. Automated Testing

1. Introduction

`Blueprint Finance` engaged Halborn to conduct a security assessment of a set of changes in their vault program from March 2nd, 2026 to March 6th, 2026. The security assessment was scoped to the smart contracts provided in the GitHub repository `glow-v1`; commit hashes and further details can be found in the Scope section of this report.

The `vault` program was upgraded using feature flags to add new withdrawal and share-delivery modes while keeping existing user flows non-breaking. The changes enable transferable shares (minted directly to user wallets, only when performance fees are disabled), optional time-locked share delivery to enforce minimum holding/transfer restrictions, an epoch-based redemption alternative to the timer-based model, and an optional redemption mode that freezes the exchange rate at request time instead of execution.

2. Assessment Summary

Halborn was provided 5 days for the engagement and assigned 1 full-time security engineer to review the security of the Solana Program in scope. The engineer is blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Solana Program.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were addressed by the `Blueprint team`. The main ones were the following:

- `Enforce in configure_vault a mandatory post-condition: whenever EPOCH_WITHDRAWALS is disabled, the stored vault.withdrawal_waiting_period must be interpreted as seconds.`
- `Enforce deposit_limit against the total share supply rather than deposit_shares.`
- `Enforce symmetry between reservation and release by using a single eligibility predicate across the lifecycle.`
- `Clarify and enforce the intended unit of the cap.`
- `Enforce a minimum-mint invariant by rejecting deposits that would mint zero shares.`

3. Test Approach And Methodology

Halborn performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors.
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities (`cargo audit`).
- Local runtime testing (`anchor test`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: glow-v1

(b) Assessed Commit ID: 273aea0

(c) Items in scope:

- programs/vault/src/events.rs
- programs/vault/src/instructions/admin/accrue_performance_fees.rs
- programs/vault/src/instructions/admin/configure_vault.rs
- programs/vault/src/instructions/admin/mod.rs
- programs/vault/src/instructions/admin/update_share_token_metadata.rs
- programs/vault/src/instructions/operator/close_operator_margin_account.rs
- programs/vault/src/instructions/user/cancel_transferable_pending_withdrawal.rs
- programs/vault/src/instructions/user/cancel_vault_pending_withdrawal.rs
- programs/vault/src/instructions/user/claim_deposited_shares.rs
- programs/vault/src/instructions/user/deposit.rs
- programs/vault/src/instructions/user/deposit_transferable.rs
- programs/vault/src/instructions/user/execute_transferable_withdrawal.rs
- programs/vault/src/instructions/user/execute_vault_withdrawal.rs
- programs/vault/src/instructions/user/initiate_transferable_withdrawal.rs
- programs/vault/src/instructions/user/initiate_withdrawal.rs
- programs/vault/src/instructions/user/mod.rs
- programs/vault/src/lib.rs
- programs/vault/src/seeds.rs
- programs/vault/src/state/mod.rs
- programs/vault/src/state/pending_deposits.rs
- programs/vault/src/state/vault.rs

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 52d7f2e
- a1aac99
- 4c4205a
- 21e56f7
- 884f3d6
- 2f7173c
- d0f046f
- cf22c1c
- 5ff521c

- 909d574

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	1	4

INFORMATIONAL
5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
EPOCH-TO-ROLLING TRANSITION CAN INTERPRET EPOCH ANCHOR AS WAITING PERIOD	MEDIUM	SOLVED - 03/23/2026
DEPOSIT LIMIT ENFORCEMENT IGNORES RESERVED SHARES UNDER FREEZE_WITHDRAWAL_YIELD, ALLOWING TOTAL SUPPLY TO EXCEED THE CAP	LOW	SOLVED - 03/25/2026
MISSING FREEZE_YIELD_ACTIVATION_TS CHECK ON INITIATE MAY LEAD TO RESERVATION LEAK	LOW	SOLVED - 03/23/2026
DEPOSIT LIMIT IS SHARE-DENOMINATED, ALLOWING TOKEN AUM TO EXCEED EXPECTED CAPACITY AS THE VAULT APPRECIATES	LOW	SOLVED - 03/25/2026
TOKEN↔SHARE CONVERSIONS CAN ROUND TO ZERO WITHOUT GUARDS	LOW	SOLVED - 03/24/2026
PENDINGDEPOSITS ACCOUNTS LACKS A CLOSE PATH PREVENTING RENT RECLAMATION AFTER TIME-LOCKED DEPOSITS	INFORMATIONAL	SOLVED - 03/25/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING VALIDATIONS IN UPDATE SHARE TOKEN METADATA	INFORMATIONAL	SOLVED - 03/25/2026
DEPOSIT EVENT DOUBLE-COUNTS TOTAL_USER_SHARES ON NON-LOCKED TRANSFERABLE DEPOSITS	INFORMATIONAL	SOLVED - 03/25/2026
UNNECESSARY MUT ACCOUNT DECLARATIONS EXPAND WRITE SURFACE	INFORMATIONAL	SOLVED - 03/25/2026
FREEZE_WITHDRAWAL_YIELD CAN CLAMP DEPOSIT_TOKENS TO ZERO, CAUSING DEPOSIT DOS AND ZERO-PRICED LIVE WITHDRAWALS	INFORMATIONAL	SOLVED - 03/25/2026

7. FINDINGS & TECH DETAILS

7.1 EPOCH-TO-ROLLING TRANSITION CAN INTERPRET EPOCH ANCHOR AS WAITING PERIOD

// MEDIUM

Description

The vault supports two withdrawal regimes: an epoch-based mode where withdrawals settle against an epoch anchor, and a rolling mode where withdrawals are gated by a `withdrawal_waiting_period` interpreted as a duration in seconds.

When the vault is switched from epoch mode to rolling mode, the transition does not guarantee that `vault.withdrawal_waiting_period` is rewritten into a valid duration. As a result, the vault can carry forward an epoch-era “settlement anchor” Unix timestamp which, once epoch mode is disabled, is subsequently treated as a waiting period in seconds, producing multi-decade lockups for new withdrawals.

This occurs because `configure_vault` only updates and bounds-checks `withdrawal_waiting_period` when `config.withdrawal_waiting_period` is explicitly provided; the max-range validation is evaluated only inside that `Some(...)` branch.

If an admin clears the `EPOCH_WITHDRAWALS` flag via `vault_flags` but omits a new waiting period in the same call, the stored field is neither updated nor revalidated, leaving the vault in an inconsistent state: epoch withdrawals disabled while `withdrawal_waiting_period` still contains the old epoch anchor.

Copy Code

```
376  if let Some(vault_flags) = config.vault_flags {
377          let vault_flags = VaultFeatureFlags::from_bits(vault_flags)
378              .ok_or(crate::ErrorCode::InvalidFeatureFlags)?;
379
380          // SHARES_TRANSFERABLE can only be changed before any shares are minted
381          let transferable_changed = vault_flags.contains(VaultFeatureFlags::SHARES_TRANSFERABLE
382              != v.flags.contains(VaultFeatureFlags::SHARES_TRANSFERABLE);
383          if transferable_changed {
384              require!(
385                  v.deposit_shares == 0,
386                  crate::ErrorCode::VaultPermissionDenied
387              );
388          }
389
390          v.flags = vault_flags;
391      }
```

Downstream, `initiate_vault_withdrawal` and `initiate_transferable_vault_withdrawal` select the rolling path whenever epoch mode is off, and fall back to `u32::try_from(vault_ref.withdrawal_waiting_period).unwrap_or(u32::MAX)` for the waiting period.

A stale anchor timestamp fits into `u32`, so the derived waiting period becomes ~1.7×10^9 seconds (~55 years) and is written into `PendingWithdrawal.withdrawal_waiting_period`.

```
let vault_ref = ctx.accounts.vault.load()?;
   let waiting_period = vault_ref
       .compute_epoch_settlement_timestamp(clock.unix_timestamp)
       .map(|settle_ts| {
           u32::try_from((settle_ts - clock.unix_timestamp).max(0)).unwrap_or(u32::MAX)
       })
       .unwrap_or(u32::try_from(vault_ref.withdrawal_waiting_period).unwrap_or(u32::MAX));
   drop(vault_ref);
```

Execution and cancellation handlers then enforce unlock eligibility via `timestamp >= withdrawal_request_timestamp + waiting_period`, meaning withdrawals initiated after the mode switch can be effectively frozen for decades with shares locked in pending-withdrawal custody.

```
let waiting_period = pending.withdrawal_waiting_period as i64;
  require!(
      timestamp >= pending.withdrawal_request_timestamp + waiting_period,
      crate::ErrorCode::WithdrawWaitingPeriodNotPassed
  );
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:M/D:N/Y:N (5.6)

Recommendation

Consider to enforce in `configure_vault` a mandatory post-condition: whenever `EPOCH_WITHDRAWALS` is **disabled**, the stored `vault.withdrawal_waiting_period` must be interpreted as seconds and therefore must fall within `[0, VAULT_WITHDRAWAL_WAITING_PERIOD_MAX_SECONDS]`; otherwise, reject the configuration change with a dedicated error.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by adding a permanent post-condition check at the end of `configure_vault` that validates `withdrawal_waiting_period` is within `[0, VAULT_WITHDRAWAL_WAITING_PERIOD_MAX_SECONDS]` whenever `EPOCH_WITHDRAWALS` is disabled in the final vault state, regardless of whether the waiting period was explicitly provided in the configuration call.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/52d7f2e78007cae57c80ecfe6d24f66de39316cd

7.2 DEPOSIT LIMIT ENFORCEMENT IGNORES RESERVED SHARES UNDER FREEZE_WITHDRAWAL_YIELD, ALLOWING TOTAL SUPPLY TO EXCEED THE CAP

// LOW

Description

When `FREEZE_WITHDRAWAL_YIELD` is enabled, frozen pending withdrawals reserve shares in `reserved_withdrawal_shares`. The vault then recomputes its depositor-facing share base as `deposit_shares = share_mint.supply - reserved_withdrawal_shares` in `update_vault()`, excluding reserved shares from the "active" share count. In `increase_vault()`, the deposit cap is enforced using this reduced base (`require!(deposit_limit >= deposit_shares)`), rather than the total share mint supply.

As a result, deposits can grow the **total** share supply up to `deposit_limit + reserved_withdrawal_shares`, because the check effectively enforces `share_mint.supply - reserved_withdrawal_shares <= deposit_limit`. This becomes incorrect once pending redemptions are cancelled.

Since `cancel_transferable_pending_withdrawal` decreases `reserved_withdrawal_shares` while the shares remain in circulation, canceling previously reserved withdrawals can immediately convert "reserved" shares back into active shares, leaving the vault with a total supply that exceeds `deposit_limit` even though the cap was never violated under the active-share accounting.

This breaks the intended invariant that `deposit_limit` bounds the maximum share supply and therefore the maximum vault capacity. The issue has been confirmed as unintended behavior by the developer given the cancelability of pending redemptions.

Copy Code

```
// Update token balances
        // Exclude reserved amounts so that frozen pending withdrawals are not counted
        // in the exchange rate seen by remaining depositors.
        self.deposit_shares = share_mint
            .supply
            .saturating_sub(self.reserved_withdrawal_shares);

```

Copy Code

```
let (signer_seeds, deposit_shares) = {
        let vault = &mut ctx.accounts.vault.load_mut()?;

        require!(
            deposit_tokens >= vault.minimum_deposit,
            crate::ErrorCode::DepositBelowMinimum
        );

        let deposit_shares = vault.increase_vault(deposit_tokens, clock.unix_timestamp)?;

        (vault.signer_seeds_owned(), deposit_shares)
    };
```

```
/// Increate the vault by depositing tokens into it, returning the number of shares to mint.
    pub fn increase_vault(&mut self, tokens: u64, timestamp: i64) -> Result<u64> {
        self.accrue_management_fees(timestamp)?;
        let shares = self.tokens_to_shares(tokens, timestamp)?;
        self.deposit_tokens = self
            .deposit_tokens
            .checked_add(tokens)
            .ok_or(crate::ErrorCode::Overflow)?;
        self.deposit_shares = self
            .deposit_shares
            .checked_add(shares)
            .ok_or(crate::ErrorCode::Overflow)?;

        // Checks
        require!(
            self.deposit_limit >= self.deposit_shares,
            crate::ErrorCode::DepositLimitReached
        );

        Ok(shares)
```

```
if is_frozen {
        let vault = &mut ctx.accounts.vault.load_mut()?;
        vault.reserved_withdrawal_tokens = vault
            .reserved_withdrawal_tokens
            .checked_sub(locked_tokens)
            .ok_or(crate::ErrorCode::Underflow)?;
        vault.reserved_withdrawal_shares = vault
            .reserved_withdrawal_shares
            .checked_sub(refund_shares)
            .ok_or(crate::ErrorCode::Underflow)?;
    }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:M/Y:N (4.2)

Recommendation

It is recommended to enforce `deposit_limit` against the total share supply rather than `deposit_shares`.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by implementing the recommendation.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/a1aac99587489b9f57410846f0ef00a1b7ebdbcd

7.3 MISSING FREEZE_YIELD_ACTIVATION_TS CHECK ON INITIATE MAY LEAD TO RESERVATION LEAK

// LOW

Description

The transferable vault implements a `FREEZE_WITHDRAWAL_YIELD` mode intended to prevent yield accrual from being attributed to assets that have entered the withdrawal pipeline.

When the flag is active, `initiate_transferable_vault_withdrawal` reserves the requested amount by incrementing `reserved_withdrawal_tokens` and `reserved_withdrawal_shares`, and records the reserved payout amount in `pending_assets`; `execute` / `cancel` are then expected to release those reservations when the withdrawal is completed or cancelled.

However, the reservation and release conditions are not symmetric. Initiation reserves purely based on the current vault flag, whereas `execute_transferable_withdrawal` and the cancellation handlers only decrement the reserved counters when:

- `pending_assets` is greater than `0`
- `pending.withdrawal_request_timestamp` is equal or greater than `FREEZE_YIELD_ACTIVATION_TS`.

This timestamp gate effectively treats withdrawals requested before `FREEZE_YIELD_ACTIVATION_TS` as “legacy” and not frozen, even if they were initiated while the flag was enabled and therefore did reserve assets. As a result, withdrawals can enter a state where they increment reserved counters at initiation but are never eligible to release those reservations on execute/cancel.

programs/vault/src/instructions/initiate_transferable_withdrawal.rs

Copy Code

```
119     let is_frozen = vault.freeze_withdrawal_yield();
```

Copy Code

```
146 let pending_assets = if is_frozen && tokens_to_withdraw > 0 {
147         let vault = &mut ctx.accounts.vault.load_mut()?;
148         vault.reserved_withdrawal_tokens = vault
149             .reserved_withdrawal_tokens
150             .checked_add(tokens_to_withdraw)
151             .ok_or(crate::ErrorCode::Overflow)?;
152         vault.reserved_withdrawal_shares = vault
153             .reserved_withdrawal_shares
154             .checked_add(shares)
155             .ok_or(crate::ErrorCode::Overflow)?;
156         tokens_to_withdraw
157     } else {
158         0
159     };
160
161     let pending = PendingWithdrawal {
162         pending_assets,
163         pending_shares: shares,
164         withdrawal_request_timestamp: clock.unix_timestamp,
165         withdrawal_waiting_period: waiting_period,
166     };
```

programs/vault/src/instructions/execute_transferable_withdrawal.rs

```
let frozen_payout = pending.pending_assets;
    let is_frozen = frozen_payout > 0
        && pending.withdrawal_request_timestamp >= crate::state::vault::FREEZE_YIELD_ACTIVATIO

    let gross_withdrawal_tokens = if is_frozen {
        // Frozen path: use the payout locked at initiation; release the reservation.
        let vault = &mut ctx.accounts.vault.load_mut()?;
        vault.reserved_withdrawal_tokens = vault
            .reserved_withdrawal_tokens
            .checked_sub(frozen_payout)
            .ok_or(crate::ErrorCode::Underflow)?;
        vault.reserved_withdrawal_shares = vault
            .reserved_withdrawal_shares
            .checked_sub(shares)
            .ok_or(crate::ErrorCode::Underflow)?;

```

In practice, if an admin enables `FREEZE_WITHDRAWAL_YIELD` prior to the activation timestamp, any withdrawal initiated in that interval will increase `reserved_withdrawal_tokens/shares` and set non-zero `pending_assets`, but when later executed or cancelled the "frozen" condition evaluates false due to the request timestamp being before the cutoff. The reserved amounts are therefore never decremented, creating permanent "phantom" reservations that monotonically accumulate over time.

This state pollution directly impacts vault accounting. `update_vault()` derives `deposit_tokens` and `deposit_shares` by subtracting the reserved counters from total assets and share supply; when reservations are never released, the effective base used for exchange-rate computation is artificially reduced.

That distorts the exchange rate observed by remaining depositors and, in the worst case, can drive derived values toward zero such that the vault appears to have little or no withdrawable value despite holding assets.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:H/D:H/Y:M (3.5)

Recommendation

It is recommended to enforce symmetry between reservation and release by using a single eligibility predicate across the lifecycle:

- Either apply the `FREEZE_YIELD_ACTIVATION_TS` gate at initiation (only reserve when `withdrawal_request_timestamp >= FREEZE_YIELD_ACTIVATION_TS`)
- Or remove the timestamp gate from execute/cancel so that any withdrawal that r`ecords pending_assets > 0` always decrements `reserved_withdrawal_tokens` and `reserved_withdrawal_shares` on completion/cancellation.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by adding validation to ensure that `clock.unix_timestamp` is greater than or equal to `FREEZE_YIELD_ACTIVATION_TS` during initialization.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/4c4205aa823b2d955c9b1838c47159dea8d2e63e

7.4 DEPOSIT LIMIT IS SHARE-DENOMINATED, ALLOWING TOKEN AUM TO EXCEED EXPECTED CAPACITY AS THE VAULT APPRECIATES

// LOW

Description

The vault enforces its deposit cap exclusively as a maximum share base: `increase_vault()` gates deposits with `require!(deposit_limit >= deposit_shares, DepositLimitReached)` after minting shares, and no equivalent check is applied to `deposit_tokens` or any token-denominated "assets under management" metric.

Since the vault's exchange rate is derived from `deposit_tokens / deposit_shares`, the token value represented by a fixed number of shares can increase over time as the vault accrues yield or otherwise appreciates (i.e., `deposit_tokens` increases without a proportional increase in `deposit_shares`).

Under these normal operating conditions, a configuration that an operator may intuitively interpret as a token-cap (e.g., "~10M USDC capacity") can be exceeded in token terms while remaining fully compliant in share terms. For example, if `deposit_limit` is set to 10M shares and the vault appreciates such that each share represents more tokens, the vault can end up managing >10M tokens even though new deposits are blocked once the share limit is reached.

This is a implementation mismatch risk: the deposit cap constrains share supply rather than token AUM, and the code does not document that distinction, making misconfiguration plausible, especially given that analogous limits in other components are token-denominated and enforced against `deposit_tokens`.

programs/vault/src/state/vault.rs

Copy Code

```rust
/// Increate the vault by depositing tokens into it, returning the number of shares to mint.
    pub fn increase_vault(&mut self, tokens: u64, timestamp: i64) -> Result<u64> {
        self.accrue_management_fees(timestamp)?;
        let shares = self.tokens_to_shares(tokens, timestamp)?;
        self.deposit_tokens = self
            .deposit_tokens
            .checked_add(tokens)
            .ok_or(crate::ErrorCode::Overflow)?;
        self.deposit_shares = self
            .deposit_shares
            .checked_add(shares)
            .ok_or(crate::ErrorCode::Overflow)?;

        // Checks
        require!(
            self.deposit_limit >= self.deposit_shares,
            crate::ErrorCode::DepositLimitReached
        );

        Ok(shares)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:L/Y:N (3.1)

Recommendation

It is recommended to clarify and enforce the intended unit of the cap.

- If the product intent is to cap shares, explicitly document `deposit_limit` as a share-supply limit (including operator guidance that token AUM may exceed the cap as the vault appreciates).
- If the intent is to cap token AUM, add an explicit token-denominated limit (e.g., deposit_limit_tokens) enforced against deposit_tokens (or an equivalent total-token metric) in the deposit path, or otherwise redesign the cap so it remains stable in token terms as the exchange rate changes.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by adding documentation comments to `deposit_limit` in `vault.rs` and `configure_vault.rs,` which explicitly clarify that the limit is expressed in shares and is not an assets under management (AUM) limit in tokens.

This is due to using shares for the deposit limit would indeed lead to oversubscription, however it is more predictable than token limits, as a vault whose limit is denominated in tokens could fluctuate between being limited vs not based on operator performance. If there are large fluctuations in performance when a vault is at its deposit limit, it could mean that additional deposits are possible if the performance declines, and vice versa.

The team preferred the more deterministic option of using shares, as they do not have this issue.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/21e56f78df8603f6f9600152594d5d208c79b6fa

7.5 TOKEN↔SHARE CONVERSIONS CAN ROUND TO ZERO WITHOUT GUARDS

// LOW

Description

The vault converts between tokens and shares using a token↔share exchange rate with **round-down** semantics in `Number128` (conversion via `.as_u64(0)`), meaning small amounts under certain exchange rates can legitimately truncate to `0`. This becomes user-impacting because two irreversible flows rely on these conversions without enforcing that the computed result is strictly positive.

On the deposit path (`deposit` and `deposit_transferable`), `increase_vault()` computes `deposit_shares` via `tokens_to_shares(tokens)`. If the exchange rate is sufficiently high, a small but non-zero token deposit can compute `deposit_shares = 0`. Neither `increase_vault()` nor the deposit handlers reject this case: the program transfers the user's tokens into the vault reserve and mints **0 shares**, effectively donating the deposit to existing shareholders and causing the depositor to lose the deposited value.

On the withdrawal initiation path (`initiate_vault_withdrawal` and `initiate_transferable_vault_withdrawal`), the program computes `tokens_to_withdraw` via `shares_to_tokens(shares)`. If the exchange rate is sufficiently low or the share amount is small, this can yield `tokens_to_withdraw = 0`.

The handlers require `shares > 0` but do not require `tokens_to_withdraw > 0`, so they can create a pending withdrawal and move the user's shares into custody even though the computed payout is zero. On execution, those shares may be burned and the user can receive **0 tokens**, depending on the live-rate path or the fixed payout recorded, resulting in a withdrawal that destroys share value without delivering underlying.

Copy Code

```
let deposit_shares = vault.increase_vault(deposit_tokens, clock.unix_timestamp)?;

        (vault.signer_seeds_owned(), deposit_shares)
    };

    // Transfer underlying from user to vault reserve
    let decimals = ctx.accounts.underlying_mint.decimals;

    let underlying_mint = ctx.accounts.underlying_mint.to_account_info();
    let mint_token_program = ctx.accounts.underlying_mint_token_program.to_account_info();
    deny_transfer_fees(&underlying_mint, &mint_token_program)?;

    tokens::transfer_tokens(
        mint_token_program,
        underlying_mint,
        ctx.accounts
            .depositor_underlying_token_account
            .to_account_info(),
        ctx.accounts.vault_reserve.to_account_info(),
        ctx.accounts.depositor.to_account_info(),
        decimals,
        deposit_tokens,


```

```
            None,
        )?;
```

Copy Code

```
let slot = pending
            .empty_deposit_slot()
            .ok_or(crate::ErrorCode::TooManyPendingDeposits)?;
        pending.populate_deposit_slot(
            slot,
            PendingDeposit {
                pending_shares: deposit_shares,
                deposit_timestamp: clock.unix_timestamp,
                delivery_waiting_period: u32::try_from(vault_data.deposit_delivery_lock_period
                    .map_err(|_| crate::ErrorCode::Overflow)?,
                redemption_waiting_period: u32::try_from(vault_data.deposit_redemption_lock_pe
                    .map_err(|_| crate::ErrorCode::Overflow)?,
            },
        )?;

        custody_info
    } else {
        ctx.accounts
            .share_token_account
            .as_ref()
            .ok_or(crate::ErrorCode::InvalidAccount)?
            .to_account_info()
    };

    tokens::mint_tokens(
        share_token_program,
        ctx.accounts.share_mint.to_account_info(),
        share_destination.clone(),
        ctx.accounts.vault.to_account_info(),
        Some(&seeds),
        deposit_shares,
    )?;
```

This is a single systemic pattern—round-down conversion to zero combined with missing post-conversion validation—that can lead to irreversible actions with zero economic output for the user (tokens transferred with no shares minted, or shares locked/burned for no tokens).

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (2.5)

Recommendation

Consider to enforce a minimum-mint invariant by rejecting deposits that would mint zero shares. For it, it is recommended to add a `require!(shares > 0, …)` inside `increase_vault` immediately after `tokens_to_shares`, before mutating state or returning.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by adding a validation to ensure the `shares` in `increase_vault` and `tokens_to_withdraw` in

- `initiate_withdrawal.rs`
- `initiate_transferable_withdrawal.rs`

are greater than 0.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/884f3d619f53656089b3ad71fd956cbf7722a61c

7.6 PENDINGDEPOSITS ACCOUNTS LACKS A CLOSE PATH PREVENTING RENT RECLAMATION AFTER TIME-LOCKED DEPOSITS

// INFORMATIONAL

Description

Transferable vaults support time-locked deposits by escrowing newly minted shares in a vault-owned custody account and tracking them in a per-depositor `PendingDeposits` PDA created via `init_if_needed` in `deposit_to_transferable_vault` with rent paid by the transaction payer.

As shares are unlocked, `claim_deposited_shares` clears the corresponding entries via `make_empty(...)` and reduces `total_pending_shares` until the account is fully emptied (`total_pending_shares == 0` and all slots cleared).

Unlike the withdrawal flow—which exposes `close_pending_withdrawal` to close an empty `PendingWithdrawals` PDA and refund its lamports—there is no equivalent instruction to close `PendingDeposits`.

The PDA can be created and mutated but not closed, even when it becomes empty, leaving the rent-exempt balance unrecoverable for users who complete the intended deposit-and-claim lifecycle.

Copy Code

```
137
138     ctx.accounts.pending_deposits.make_empty(index)?;
139
140     emit!(events::ClaimDepositedShares {
141         depositor: ctx.accounts.depositor.key(),
142         vault: ctx.accounts.vault.key(),
143         share_mint: ctx.accounts.share_mint.key(),
144         shares_claimed: shares,
145         deposit_index,
146     });
```

While the per-account amount is small, the effect is permanent per depositor and introduces an avoidable lifecycle asymmetry between deposit and withdrawal tracking accounts.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:N/D:L/Y:N (1.3)

Recommendation

It is recommended to add a `close_pending_deposits` instruction analogous to `close_pending_withdrawal`, gated on the account being empty.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by implementing an instruction to close the pending deposits.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/2f7173c67ff046b485eec17e647e817d3bb422c9

7.7 MISSING VALIDATIONS IN UPDATE SHARE TOKEN METADATA

// INFORMATIONAL

Description

The vault's admin-only `update_share_token_metadata` instruction allows the vault authority to create or update Metaplex Token Metadata for the vault share mint.

Metadata Fields

In addition, the instruction accepts optional metadata fields (`name`, `symbol`, `uri`) and forwards them into Metaplex create/update CPIs (create uses defaults only when the option is `None`, but `Some("")` propagates an empty string; update merges caller-provided values over existing metadata). The handler does not enforce non-emptiness for provided strings nor enforce the Metaplex length limits (`MAX_NAME_LENGTH`, `MAX_SYMBOL_LENGTH`, `MAX_URI_LENGTH`).

As a result, empty or oversize values can be written (on update) or can cause Metaplex CPI failures (on create/update), producing avoidable DoS-style failures, wasted fees, and ambiguous or misleading on-chain metadata. The inconsistency is amplified by the fact that the program already relies on Metaplex length constants when parsing existing metadata, but does not apply equivalent bounds to incoming parameters.

programs/vault/src/instructions/update_share_token_metadata

Copy Code

```
pub fn update_share_token_metadata_handler(
    ctx: Context<UpdateShareTokenMetadata>,
    metadata: ShareTokenMetadata,
) -> Result<()> {
    let vault = ctx.accounts.vault.load()?;

    // Explicit authorization check (has_one constraint should handle this, but being explicit
    require!(
        vault.authority == ctx.accounts.authority.key(),
        crate::ErrorCode::UnauthorizedInvocation
    );

    let share_mint = ctx.accounts.share_mint.key();

    // Note: The metadata PDA is validated by Anchor's seeds constraint above,
    // which efficiently derives and validates the PDA without using find_program_address.

    // Check if metadata account exists
    let metadata_account_info = ctx.accounts.metadata.to_account_info();
    let metadata_exists = metadata_account_info.data_len() > 0;

    if !metadata_exists {
        // Create metadata account
        let name = metadata.name.unwrap_or_else(|| "Vault Share".to_string());
        let symbol = metadata.symbol.unwrap_or_else(|| "VAULT".to_string());
        let uri = metadata.uri.unwrap_or_default();
```

Copy Code

```
(
                metadata.name.unwrap_or(existing_name),
                metadata.symbol.unwrap_or(existing_symbol),
```

```
                metadata.uri.unwrap_or(existing_uri),
            )
```

Sysvar Instructions

On the update path (when the metadata PDA already exists), the handler performs a CPI to Metaplex `UpdateV1` and includes a caller-supplied `sysvar_instructions` account as an `UncheckedAccount` without constraining it to the canonical Instructions sysvar.

Although the instruction data references the expected sysvar ID, the CPI account list passes whatever account the caller provides, meaning the Metaplex program will read "transaction instructions" from potentially non-sysvar, attacker-controlled data. This violates least-privilege account validation and widens the trust boundary of a security-sensitive CPI dependency.

programs/vault/src/instructions/update_share_token_metadata

Copy Code

```
    pub token_program: Interface<'info, TokenInterface>,

    /// CHECK: Metaplex Token Metadata program
    #[account(address = TOKEN_METADATA_PROGRAM_ID)]
    pub token_metadata_program: UncheckedAccount<'info>,

    pub system_program: Program<'info, System>,

    /// CHECK: Sysvar instructions account for Metaplex
    pub sysvar_instructions: UncheckedAccount<'info>,
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

It is recommended to:

- Validate `name`, `symbol`, and `uri` before building CPIs to prevent assigning invalid values.
- Constrain `sysvar_instructions` to the canonical Instructions sysvar via an explicit address constraint

Remediation Comment

SOLVED: The `Blueprint` team solved this issue by implementing the suggested validations.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/d0f046ffa27a2cbe2f9b6c461d8e6e26c47ad194

7.8 DEPOSIT EVENT DOUBLE-COUNTS TOTAL_USER_SHARES ON NON-LOCKED TRANSFERABLE DEPOSITS

// INFORMATIONAL

Description

The vault supports transferable deposits where newly minted shares are delivered either directly to the depositor's share token account (non-locked path) or escrowed in a vault-owned custody account during a time lock (locked path).

In both cases the handler emits a `Deposit` event whose `total_user_shares` field is intended to represent the depositor's post-deposit share balance in the destination account that received the mint.

On the non-locked path, the program mints `deposit_shares` to the depositor's share token account via CPI. After the CPI returns, `share_ta.amount` already reflects the post-mint balance.

The handler then computes `total_user_shares` as `share_ta.amount.checked_add(deposit_shares)`, effectively adding the minted shares a second time. This causes the event to emit an inflated value equal to `previous_balance + 2 * deposit_shares` rather than the true post-mint balance.

programs/vault/src/instructions/deposit_transferable.rs

Copy Code

```
281  tokens::mint_tokens(
282          share_token_program,
283          ctx.accounts.share_mint.to_account_info(),
284          share_destination.clone(),
285          ctx.accounts.vault.to_account_info(),
286          Some(&seeds),
287          deposit_shares,
288      )?;
289
290      // For margin account position tracking (non-locked path only, since locked
291      // deposits don't give the margin account shares immediately)
292      let depositor_is_margin = is_margin_account(&ctx.accounts.depositor, signer_seeds.airspace
293
294      // total_user_shares = user's share balance in their wallet after this deposit.
295      // For locked deposits, shares are escrowed in custody so the user's wallet has 0.
296
297  let total_user_shares = if !is_locked {
298          let share_ta = ctx
299              .accounts
300              .share_token_account
301              .as_ref()
302              .ok_or(crate::ErrorCode::InvalidAccount)?;
303          share_ta
304              .amount
305              .checked_add(deposit_shares)
306              .ok_or(crate::ErrorCode::Overflow)?
307      } else {
308          0u64
309      };
```

Copy Code

```
344  emit!(Deposit {
345          payer: ctx.accounts.payer.key(),
```

```
            depositor: ctx.accounts.depositor.key(),
            vault: ctx.accounts.vault.key(),
            underlying_mint: ctx.accounts.underlying_mint.key(),
            underlying_mint_decimals: decimals,
            depositor_underlying_token_account: ctx.accounts.depositor_underlying_token_account.ke
            share_token_account: share_destination.key(),
            vault_user: Pubkey::default(),
            deposit_tokens,
            deposit_shares,
            total_user_shares,
        });
```

This can affect only off-chain consumers: no on-chain accounting or transfer logic relies on `total_user_shares` from the event, and the locked path correctly emits `total_user_shares = 0` because shares are escrowed.

However, indexers, dashboards, and analytics that interpret `total_user_shares` as the user's actual wallet balance will over-report share holdings by `deposit_shares` per non-locked deposit, leading to incorrect reporting until corrected.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.8)

Recommendation

It is recommended to compute `total_user_shares` from the correct balance snapshot

- either capture the pre-mint balance and add `deposit_shares`,
- or simply use the post-mint `share_ta.amount` as-is after the CPI.

Remediation Comment

SOLVED: The `Blueprint` solved this issue by replacing the cached calculation of the sum, with an explicit `share_ta.reload()` followed by a direct read of `share_ta.amount`. This ensures the emitted `total_user_shares` value in the Deposit event is read directly from the on-chain account state after the mint CPI.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/cf22c1c1198b53031889ad27a9b40800ca239440

7.9 UNNECESSARY MUT ACCOUNT DECLARATIONS EXPAND WRITE SURFACE

// INFORMATIONAL

Description

Several vault instructions declare accounts as mutable even though they are never written, expanding the programs effective write surface beyond what the handlers require.

Underlying Mint Account

In `execute_transferable_withdrawal` and `execute_vault_withdrawal`, the `underlying_mint` account is marked `#[account(mut)]` despite being used strictly read-only: the instruction only reads `decimals`, passes the mint into `deny_transfer_fees()` (read-only validation), and into `transfer_tokens()` via `TransferChecked`.

No mint/burn is performed on `underlying_mint`; the only token state transitions are burning share tokens from the pending-withdrawal custody (share mint) and transferring underlying from `vault_reserve` to the destination. The mint's correctness is already implicitly bound through the vault PDA derivation, so mutability is not functionally required.

Copy Code

```
67  #[account(
68          mut,
69          seeds = [VAULT_DEPOSIT_MINT_SEED, vault.key().as_ref()],
70          bump,
71      )]
72      pub share_mint: Box<InterfaceAccount<'info, Mint>>,
73
74      #[account(mut)]
75      pub underlying_mint: Box<InterfaceAccount<'info, Mint>>,
```

Vault Account

A situation exists in `update_share_token_metadata`. The `vault` account is declared `#[account(mut)]` even though the instruction never mutates vault state; it is loaded to validate authority, derive signer seeds, and is passed as mint/update authority in Metaplex Token Metadata CPIs, while the only on-chain state change occurs in the metadata account owned by the token-metadata program.

Additionally, the vault account in this context is not constrained by PDA seeds, making the "mutable vault" declaration both unnecessary and misleading with respect to what can be modified.

Copy Code

```
34  pub struct UpdateShareTokenMetadata<'info> {
35      #[account(mut)]
36      pub authority: Signer<'info>,
37
38
```

```
39      #[account(mut)]
        pub vault: AccountLoader<'info, Vault>,
```

While this does not create an immediate exploit on its own, it weakens defense-in-depth and auditability: unnecessary `mut` enlarges the set of accounts the program is permitted to write, increases the blast radius of any future handler changes, and can obscure intent for reviewers and integrators.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove `#[account(mut)]`

- from `underlying_mint` in both withdrawal instructions
- from `vault` in `update_share_token_metadata`

Consider to add explicit account constraints (e.g., vault PDA seeds in `update_share_token_metadata`) so derivation and roles are unambiguous and enforced by the account context.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by implementing the recommendation.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/5ff521c39bc06adbb9d3f0ede23a108528df10b2

7.10 FREEZE_WITHDRAWAL_YIELD CAN CLAMP DEPOSIT_TOKENS TO ZERO, CAUSING DEPOSIT DOS AND ZERO-PRICED LIVE WITHDRAWALS

// INFORMATIONAL

Description

The transferable vault supports a `FREEZE_WITHDRAWAL_YIELD` mode in which a withdrawal's payout is fixed at initiation and the vault books the locked amounts into `reserved_withdrawal_tokens` and `reserved_withdrawal_shares`.

These reservations are excluded from depositor-facing accounting: `update_vault` recomputes `deposit_tokens` and `deposit_shares` from on-chain balances and operator valuation while subtracting the reserved amounts, so that the exchange rate applied to remaining depositors does not include assets already earmarked for pending withdrawals.

However, the vault can enter a pathological "zero-rate" state when accumulated reservations exceed the vault's current asset base (e.g., withdrawals were frozen at a high operator valuation and the operator later loses value, or the reserve is drawn down). In `update_vault`, `deposit_tokens` is computed as `total_tokens.saturating_sub(reserved_withdrawal_tokens)`.

If `reserved_withdrawal_tokens >= total_tokens`, the saturating subtraction silently clamps `deposit_tokens` to `0` rather than rejecting the state as invalid. With `deposit_shares` still above `minimum_shares_dust_threshold`, the vault state (now with `deposit_tokens = 0`) yields a zero token→share exchange rate whenever `token_to_share_exchange_rate()` is derived (`0 / deposit_shares`).

Once the vault yields a zero token→share rate, downstream flows that assume a positive rate begin to fail or misbehave. Deposit handlers (`deposit_to_transferable_vault_handler` / `deposit_to_vault_handler`) do not refresh valuation before crediting deposits; they call `vault.increase_vault(...)`, which converts tokens to shares via `tokens_to_shares` by dividing by the derived exchange rate.

When the rate is zero, this triggers a `ZeroDivision` in `Number128` arithmetic and the deposit transaction reverts, effectively halting new deposits until the state is corrected. In parallel, any non-frozen withdrawal execution path that prices shares using the live exchange rate (e.g., `shares_to_tokens` inside `decrease_vault`) can compute a zero token payout under the same conditions, meaning "live" withdrawals may complete while delivering no underlying, whereas frozen withdrawals remain unaffected because they redeem against their pre-recorded locked payout.

programs/vault/src/state/vault.rs

Copy Code

```
376         // Update token balances
377             // Exclude reserved amounts so that frozen pending withdrawals are not counted
378             // in the exchange rate seen by remaining depositors.
379             self.deposit_shares = share_mint
380
```

```
            .supply
            .saturating_sub(self.reserved_withdrawal_shares);
        // Deposit tokens are the sum of idle tokens, operator tokens
        // The vault reserve balance already includes any unwithdrawn uncollected fees, so we
        // do not include them again.
        let total_tokens = operator_tokens
            .checked_add(vault_reserve.amount)
            .ok_or(crate::ErrorCode::Overflow)?;
        self.deposit_tokens = total_tokens.saturating_sub(self.reserved_withdrawal_tokens);
```

Copy Code

```
    pub fn increase_vault(&mut self, tokens: u64, timestamp: i64) -> Result<u64> {
        self.accrue_management_fees(timestamp)?;
        let shares = self.tokens_to_shares(tokens, timestamp)?;
```

Copy Code

```
    pub fn tokens_to_shares(&self, tokens: u64, timestamp: i64) -> Result<u64> {
        let token_to_share_rate = self.token_to_share_exchange_rate(timestamp)?;
        Ok(Number128::from_decimal(tokens, 0)
            .safe_div(token_to_share_rate)?
            .as_u64(0))
```

Copy Code

```
    pub fn shares_to_tokens(&self, shares: u64, timestamp: i64) -> Result<u64> {
        let token_to_share_rate = self.token_to_share_exchange_rate(timestamp)?;
        Ok(token_to_share_rate
            .safe_mul(Number128::from_decimal(shares, 0))?
            .as_u64(0))
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to prevent the vault from entering a zero-rate state by rejecting the case `reserved_withdrawal_tokens > total_tokens` in `update_vault.`

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by replacing `saturating_sub` with `checked_sub` in `update_vault` for both `deposit_shares` and `deposit_tokens` computations, causing the instruction to revert with an explicit Underflow error when accumulated reservations exceed the vault's current asset base.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/909d5743fcb8f3747ee0450e2181486a0f9399f0

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was `cargo audit`, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in `https://crates.io` are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Cargo Audit Results

ID	CRATE	DESCRIPTION
RUSTSEC-2026-0007	bytes	Integer overflow in `BytesMut::reserve`
RUSTSEC-2025-0024	crossbeam-channel	crossbeam-channel: double free on Drop
RUSTSEC-2024-0344	curve25519-dalek	Timing variability in `curve25519-dalek`'s `Scalar29::sub` / `Scalar52::sub`
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
RUSTSEC-2025-0022	openssl	Use-After-Free in `Md::fetch` and `Cipher::fetch`
RUSTSEC-2025-0009	ring	Some AES functions may panic when overflow checking is enabled.
RUSTSEC-2025-0009	ring	Some AES functions may panic when overflow checking is enabled.
RUSTSEC-2026-0001	rkyv	Potential Undefined Behaviors in `Arc<T>Rc<T>` impls of `from_value` on OOM
RUSTSEC-2026-0009	time	Denial of Service via Stack Exhaustion
RUSTSEC-2025-0055	tracing-subscriber	Logging user input may result in poisoning logs with ANSI escape sequences

---

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.