// Security Assessment

03.31.2026 - 04.01.2026

Transferrable Shares Vault: Locked Withdrawal

Blueprint Finance

HALBORN

Transferrable Shares Vault: Locked Withdrawal - Blueprint Finance

Prepared by: HALBORN

Last Updated 04/17/2026

Date of Engagement: March 31st, 2026 - April 1st, 2026

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	0	0	5

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
   - 7.1 Saturating_sub use may silently mask underflow
   - 7.2 Reset_epochs can erase outstanding epoch commitments, allowing per-epoch withdrawal caps to be overfilled
   - 7.3 Missing mint::token_program constraints in cancel/execute
   - 7.4 Dead/redundant code in withdrawal and epoch tracking paths increases maintenance risk
   - 7.5 Incorrect error classification in pendingwithdrawals aggregate accounting
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 31st, 2026 to April 1st, 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 to extend transferable-share vault functionality in two key areas. It introduces a new redemption flow that allows depositors to request redemptions after the redemption lock expires but before the transfer lock period ends; implementing this required breaking changes to the PendingWithdrawal layout and therefore includes a dedicated migration instruction to update existing accounts. Additionally, it adds epoch-based redemption throttling, implemented via a separate EpochTracker PDA to avoid further breaking changes to the core vault account layout.

2. Assessment Summary

Halborn was provided 2 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 have been addressed by the `Blueprint team`. The main ones were the following:

- `Replace saturating_sub with checked_sub and return a dedicated underflow error.`
- `Call clean_expired() before executing the reset to discard already-settled slots, and verify that no remaining slot has pending_shares`
- `Add mint::token_program constraints to the corresponding mint accounts.`
- `Remove unused state/variables and redundant computations, and avoid storing fields that are already enforced by PDA derivation`
- `Return an underflow/state-integrity error when checked_sub fails.`

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: c216ec8

(c) Items in scope:

- programs/vault/src/events.rs
- programs/vault/src/instructions/admin/configure_epoch_tracker.rs
- programs/vault/src/instructions/admin/migrate_pending_withdrawals.rs
- programs/vault/src/instructions/admin/mod.rs
- programs/vault/src/instructions/margin/margin_instant_liquidation.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/execute_vault_withdrawal.rs
- programs/vault/src/instructions/user/initiate_transferable_withdrawal.rs
- programs/vault/src/instructions/user/initiate_transferable_withdrawal_from_custody.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/epoch_tracker.rs
- programs/vault/src/state/pending_deposits.rs
- programs/vault/src/state/pending_withdrawals.rs

REMEDIATION COMMIT ID:

- 2cf7998
- 7d9bb09
- b8686ad
- e205007
- 1e06d34

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW

0 0 0 0

INFORMATIONAL
5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SATURATING_SUB USE MAY SILENTLY MASK UNDERFLOW	INFORMATIONAL	SOLVED - 04/06/2026
RESET_EPOCHS CAN ERASE OUTSTANDING EPOCH COMMITMENTS, ALLOWING PER-EPOCH WITHDRAWAL CAPS TO BE OVERFILLED	INFORMATIONAL	SOLVED - 04/06/2026
MISSING MINT::TOKEN_PROGRAM CONSTRAINTS IN CANCEL/EXECUTE	INFORMATIONAL	SOLVED - 04/06/2026
DEAD/REDUNDANT CODE IN WITHDRAWAL AND EPOCH TRACKING PATHS INCREASES MAINTENANCE RISK	INFORMATIONAL	SOLVED - 04/06/2026
INCORRECT ERROR CLASSIFICATION IN PENDINGWITHDRAWALS AGGREGATE ACCOUNTING	INFORMATIONAL	SOLVED - 04/06/2026

7. FINDINGS & TECH DETAILS

7.1 SATURATING_SUB USE MAY SILENTLY MASK UNDERFLOW

// INFORMATIONAL

Description

The epoch withdrawal limiter relies on `EpochTracker` slots to track committed shares per settlement period via `EpochSlot.pending_shares`. On withdrawal initiation, `allocate()` increments `pending_shares` for the target slot and enforces the per-epoch cap by admitting requests only when `pending_shares + new_shares <= epoch_limit`.

When a user cancels a pending withdrawal, `cancel_vault_pending_withdrawal` calls `tracker.decrement(settlement_ts, shares)` to reduce the slot's committed total and restore capacity.

However, `decrement` currently applies `slot.pending_shares = slot.pending_shares.saturating_sub(shares)`. If `shares` exceeds the slot's current `pending_shares` —whether due to a prior state divergence (e.g., after an epoch reset that lost commitments) or any future accounting inconsistency—the subtraction silently clamps to zero instead of surfacing an invariant violation.

Copy Code

```
111 pub fn decrement(&mut self, settlement_ts: i64, shares: u64) {
112         for slot in self.epochs.iter_mut() {
113             if slot.settlement_ts == settlement_ts {
114                 slot.pending_shares = slot.pending_shares.saturating_sub(shares);
115                 return;
116             }
117         }
```

This is inconsistent with `allocate()`, which uses checked arithmetic and fails explicitly on overflow. A silent underflow can make an epoch slot appear empty even if withdrawals are still effectively committed for that settlement, allowing subsequent initiations to refill the slot up to `epoch_limit` and thereby weakening the intended threshold-based liquidity protection without producing an on-chain error signal.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:M/D:L/Y:N (1.9)

Recommendation

Consider to replace `saturating_sub` with `checked_sub` and return a dedicated underflow error when `shares > pending_shares`.

Remediation Comment

SOLVED: The **Blueprint** team solved this issue by implemented the suggested remediation.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/2cf79988c76c3e500af99bb1debcf93542fd0736

7.2 RESET_EPOCHS CAN ERASE OUTSTANDING EPOCH COMMITMENTS, ALLOWING PER-EPOCH WITHDRAWAL CAPS TO BE OVERFILLED

// INFORMATIONAL

Description

The vault program supports an epoch-based withdrawal mechanism intended to protect liquidity by limiting the total shares redeemable in any settlement period. When enabled, withdrawals are queued to a future settlement timestamp and the aggregate shares committed per epoch are capped by `threshold_bps`.

This cap is enforced by the `EpochTracker`, which maintains per-epoch slots tracking `pending_shares` (committed shares) and the computed `epoch_limit`, and admits new withdrawals only when `pending_shares + new_shares <= epoch_limit`.

The admin instruction `configure_epoch_tracker` includes a `reset_epochs` option intended to clear stale slots after changes to the epoch schedule. When `reset_epochs = true`, the handler zeroes all epoch slots via `tracker.reset_epochs()` but does not first reconcile state and does not verify whether any slot still represents future-settling commitments.

Consequently, the tracker can be reset while user `PendingWithdrawals` remain queued for upcoming epochs, causing the on-chain limiter to lose all accounting of already-committed shares while the underlying withdrawal queue remains intact.

After such a reset, subsequent withdrawal initiations begin filling newly “empty” epoch slots from `pending_shares = 0`, unaware of the pre-reset commitments. This allows the same settlement epoch to be filled again up to the full `epoch_limit`, so the total shares committed for that settlement (pre-reset + post-reset) can exceed the configured `threshold_bps`.

Copy Code

```
62  pub fn configure_epoch_tracker_handler(
63      ctx: Context<ConfigureEpochTracker>,
64      threshold_bps: u16,
65      reset_epochs: bool,
66  ) -> Result<()> {
67      require!(
68          threshold_bps <= 10_000,
69          crate::ErrorCode::InvalidEpochConfig
70      );
71
72      let tracker = &mut ctx.accounts.epoch_tracker;
73
74      // Set vault reference on first init
75      if tracker.vault == Pubkey::default() {
76          tracker.vault = ctx.accounts.vault.key();
77      }
78
79      tracker.threshold_bps = threshold_bps;
80
81      if reset_epochs {
82          tracker.reset_epochs();
83      }
```

```
    pub fn reset_epochs(&mut self) {
         for slot in self.epochs.iter_mut() {
             *slot = EpochSlot::default();
         }
     }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:M/Y:N (1.8)

Recommendation

To address this issue, consider to call `clean_expired()` before executing the reset to discard already-settled slots, and then verify that no remaining slot has `pending_shares > 0` before proceeding:

```
if reset_epochs {
    let now = Clock::get()?.unix_timestamp;
    tracker.clean_expired(now);
    require!(
        tracker.epochs.iter().all(|s| s.pending_shares == 0),
        crate::ErrorCode::EpochHasActivePendingShares
    );
    tracker.reset_epochs();
}
```

If resetting with active commitments is intentional, this behavior and the operator's responsibility to adjust `threshold_bps` accordingly should be explicitly documented in the instruction.

Remediation Comment

SOLVED: The **Blueprint** team solved this issue by extending the documentation to clarify the intended behavior. The client stated that It is expected that changing epoch configurations would be infrequent, and preventing a change if there is a pending redemption queue could prevent the authority from making a change for over 1 epoch, or necessitate disabling redemptions for up to an epoch. A workaround for authorities would be to calculate a suitable limit for the epoch that takes into account existing redemptions, then change that limit at the start of the next epoch.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/7d9bb09c20e69f6140d2a0c2e6f7c6eb3e10ad6d

7.3 MISSING MINT::TOKEN_PROGRAM CONSTRAINTS IN CANCEL/EXECUTE

// INFORMATIONAL

Description

The vault supports both SPL Token and Token-2022 mints through Anchor `InterfaceAccount` types. In this model, the correct way to bind a mint to its owning token program at account-validation time is Anchor's `mint::token_program` constraint, which ensures the mint is owned by the expected token program before the handler executes.

Several of the initiation instructions updated in this change set apply this correctly, like the initiate instructions, which bind `share_mint` with `mint::token_program = token_program`).

However, the corresponding cancellation and execution instructions do not consistently enforce the same invariant. In `cancel_vault_pending_withdrawal` and `cancel_transferable_pending_withdrawal`, `share_mint` is an `InterfaceAccount<Mint>` without a `mint::token_program` binding.

In `execute_vault_withdrawal`, both `share_mint` and `underlying_mint` lack the constraint, even though the relevant token program accounts (e.g., mint token program / underlying mint token program) are passed and used by the handler to perform burns and transfers.

While the handler may dispatch CPI calls using the provided token-program accounts, Anchor will not pre-validate that the supplied mints are actually owned by those programs. At best this results in confusing runtime failures; at worst it widens the trust surface and introduces inconsistency across flows, since initiation paths correctly enforce ownership while cancel/execute paths do not.

Copy Code

```
68  #[account(
69          mut,
70          seeds = [VAULT_DEPOSIT_MINT_SEED, vault.key().as_ref()],
71          bump,
72      )]
73      pub share_mint: Box<InterfaceAccount<'info, Mint>>,
74
75      #[account(
76          mut,
77          seeds = [VAULT_PENDING_WITHDRAWALS_CUSTODY_SEED, vault.key().as_ref()],
78          bump,
79      )]
80      pub vault_pending_withdrawals_custody: AccountInfo<'info>,
81
82      /// Destination for refunded shares: must be owned by the withdrawer.
83      /// Used when the delivery lock has passed or was never set (wallet redemption).
84      #[account(
85          mut,
86          token::mint = share_mint,
87          token::authority = withdrawer,
88      )]
89      pub destination_share_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
90
91      pub mint_token_program: Interface<'info, TokenInterface>,
92
93      // --- Optional accounts below (after all required accounts to preserve index compatibilit
94      /// Per-withdrawer pending deposits record. Required when the delivery lock
95      /// has not yet passed (shares must return to deposit custody).
```

```
96      #[account(
97          mut,
98          seeds = [
99              VAULT_PENDING_DEPOSITS_SEED,
100             vault.key().as_ref(),
101             withdrawer.key().as_ref(),
102         ],
103         bump,
104     )]
105     pub pending_deposits: Option<Box<Account<'info, PendingDeposits>>>,
106
107     /// Vault-owned custody for escrowed deposit shares. Required when the
108     /// delivery lock has not yet passed.
109     #[account(
110         mut,
111         seeds = [VAULT_PENDING_DEPOSITS_CUSTODY_SEED, vault.key().as_ref()],
112         bump,
113         token::mint = share_mint,
114     )]
115     pub vault_pending_deposits_custody: Option<Box<InterfaceAccount<'info, TokenAccount>>>,
116
117
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

Considere to add `mint::token_program` constraints to the corresponding mint accounts.

Remediation Comment

SOLVED: The **Blueprint** team solved this issue by adding the suggested validations.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/b8686ad81294320badb7fed1ab161f3dd717fa79

7.4 DEAD/REDUNDANT CODE IN WITHDRAWAL AND EPOCH TRACKING PATHS INCREASES MAINTENANCE RISK

// INFORMATIONAL

Description

Several code paths contain dead or redundant logic that is no longer functionally used, increasing cognitive overhead and the risk of future refactor mistakes.

`Initiate_transferable_withdrawal_from_custody`

The instruction assigns `let _airspace = vault.airspace;` immediately after loading the vault but never references it; the underscore prefix explicitly suppresses the unused-variable warning, indicating leftover dead code after a refactor, while the actual airspace check later uses `signer_seeds.airspace`.

Copy Code

```
181     let _airspace = vault.airspace;
182      let is_frozen = vault.freeze_withdrawal_yield()
183          && clock.unix_timestamp >= crate::state::vault::FREEZE_YIELD_ACTIVATION_TS;
184
185      drop(vault);
```

`Epoch_tracker.Rs`

The `EpochTracker` struct includes a `vault: Pubkey` field that is written once during initialization (set only if default) but is never read, validated, or used for authorization. The tracker-to-vault binding is already enforced by the PDA derivation (`[EPOCH_TRACKER_SEED, vault.key()]`), so the extra field provides no security or functionality while consuming 32 bytes of on-chain storage.

Additionally, `epoch_tracker.rs` contains redundant computation in the waiting-period logic: `base_waiting` is derived using `compute_epoch_settlement_timestamp` but is only returned in the non-epoch branch; when `EPOCH_WITHDRAWALS` is active (the primary use case), this value is discarded and the epoch allocation result is used instead.

The function then calls `compute_epoch_settlement_timestamp` again to compute `target_settlement`, making the first invocation unnecessary on the epoch path.

Copy Code

```
26  pub struct EpochTracker {
27      pub vault: Pubkey,
28      /// Overflow threshold in basis points of `deposit_shares` (0 = disabled).
29      pub threshold_bps: u16,
30      pub epochs: [EpochSlot; NUM_EPOCH_SLOTS],
31  }
```

While these issues are not directly exploitable, they weaken maintainability and auditability, and in the case of the unused `vault` field, impose permanent on-chain storage overhead.

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 unused state/variables and redundant computations, and avoid storing fields that are already enforced by PDA derivation. Consider also to keep the epoch/withdrawal helpers to a single source of truth per path to reduce dead code and maintenance risk.

Remediation Comment

SOLVED: The **Blueprint** team solved this issue by implementing the suggested remediation.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/e2050073310f214d634cca9e7f577783704ba9e0

7.5 INCORRECT ERROR CLASSIFICATION IN PENDINGWITHDRAWALS AGGREGATE ACCOUNTING

// INFORMATIONAL

Description

The `PendingWithdrawals` accounting model maintains two redundant representations of the same state:

- per-slot entries `withdrawals[i].pending_{assets,shares}`
- aggregate counters `total_pending_{assets,shares}`.

A core integrity invariant is that the aggregate totals must equal the sum of all non-empty slot amounts, and that clearing a slot must decrement the totals by exactly that slot's `pending_assets` and `pending_shares`.

When a pending withdrawal is cancelled or otherwise removed, `PendingWithdrawals::make_empty()` enforces this invariant by subtracting the slot's amounts from the aggregates using `checked_sub`. However, both subtraction failures (`None`, indicating an arithmetic underflow) are mapped to `ErrorCode::Overflow`:

Copy Code

```
89  pub fn make_empty(&mut self, index: usize) -> Result<()> {
90          if index >= self.withdrawals.len() {
91              return Err(crate::ErrorCode::InvalidWithdrawIndex.into());
92          }
93          let withdrawal = self
94              .withdrawals
95              .get_mut(index)
96              .ok_or(crate::ErrorCode::InvalidWithdrawIndex)?;
97
98          self.total_pending_assets = self
99              .total_pending_assets
100             .checked_sub(withdrawal.pending_assets)
101             .ok_or(crate::ErrorCode::Overflow)?;
102         self.total_pending_shares = self
103             .total_pending_shares
104             .checked_sub(withdrawal.pending_shares)
105             .ok_or(crate::ErrorCode::Overflow)?;
```

This means that if the totals ever diverge from the per-slot sums, the failure would be reported as an "overflow" despite being an underflow/invariant violation. While this does not directly enable unauthorized asset movement, it materially degrades observability and incident response.

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 return an underflow/state-integrity error when `checked_sub` fails.

Remediation Comment

SOLVED: The **Blueprint** team solved this issue by updating the men mentioned error as suggested.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/1e06d3426f4bbf4d8b3f0cf9945c76ea89f0e14f

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-2026-0037	quinn-proto	Denial of service in Quinn endpoints
RUSTSEC-2026-0067	tar	`unpack_in` can chmod arbitrary directories by following symlinks
RUSTSEC-2026-0068	tar	tar-rs incorrectly ignores PAX size headers if header size is nonzero
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.