// Security Assessment

09.29.2025 - 10.16.2025

Glow Margin Vaults

Blueprint Finance

HALBORN

Glow Margin Vaults - Blueprint Finance

Prepared by: HALBORN

Last Updated 11/26/2025

Date of Engagement: September 29th, 2025 - October 16th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	1	1	0	2	6

TABLE OF CONTENTS

8. Automated Testing

1. Introduction

The Glow team engaged Halborn to conduct a security assessment on their Glow **Vault Solana program** beginning on September 29 2025, and ending on October 16, 2025. 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 2.5 weeks for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

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

In summary, Halborn identified some opportunities to reduce the likelihood and impact of risks, and the Glow team implemented improvements to address them. The main ones were the following:

- Ensure both management and performance fees accounting is correctly handled.
- Ensure the performance fees are charged on actual profit at the time of withdrawal.
- Validate that operators have correct permissions.

The recommendation rated as low-risk to validate potentially dangerous or incompatible Token2022 extensions has been partially addressed, with a comprehensive solution already planned by the Glow team for upcoming releases.

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 (`solana-test-framework`)

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

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
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
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

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

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

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

SCOPE (S):

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

METRICS:

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

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

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

The score is rounded up to 1 decimal places.

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

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: glow-v1

(b) Assessed Commit ID: https://github.com/Blueprint-Finance/glow-v1/pull/2437/commits/afde596e321892f23f396e4887cff1c5db3b2aff

(c) Items in scope:

- Anchor.toml
- programs/margin-pool/src/instructions/withdraw.rs
- programs/margin/src/adapter.rs
- programs/margin/src/instructions.rs
- programs/margin/src/instructions/close_account.rs
- programs/margin/src/instructions/configure/configure_account_constraints.rs
- programs/margin/src/instructions/configure/mod.rs
- programs/margin/src/instructions/positions/transfer_deposit.rs
- programs/margin/src/lib.rs
- programs/margin/src/seeds.rs
- programs/margin/src/state/account.rs
- programs/margin/src/state/config.rs
- programs/vault/Cargo.toml
- programs/vault/Xargo.toml
- programs/vault/src/events.rs
- programs/vault/src/instructions.rs
- programs/vault/src/instructions/admin/accept_account_change.rs
- programs/vault/src/instructions/admin/accrue_performance_fees.rs
- programs/vault/src/instructions/admin/assign_vault_operator_admin.rs
- programs/vault/src/instructions/admin/configure_vault.rs
- programs/vault/src/instructions/admin/create_vault.rs
- programs/vault/src/instructions/admin/mod.rs
- programs/vault/src/instructions/admin/propose_account_change.rs
- programs/vault/src/instructions/operator/close_operator_margin_account.rs
- programs/vault/src/instructions/operator/create_operator_margin_account.rs
- programs/vault/src/instructions/operator/mod.rs
- programs/vault/src/instructions/operator/operator_deposit_to_vault.rs
- programs/vault/src/instructions/operator/operator_transfer_from_margin.rs
- programs/vault/src/instructions/operator/operator_transfer_to_margin.rs
- programs/vault/src/instructions/operator/operator_withdraw_from_vault.rs
- programs/vault/src/instructions/user/cancel_vault_pending_withdrawal.rs
- programs/vault/src/instructions/user/create_vault_pending_withdrawal.rs
- programs/vault/src/instructions/user/deposit.rs

- programs/vault/src/instructions/user/execute_vault_withdrawal.rs
- programs/vault/src/instructions/user/initiate_withdrawal.rs
- programs/vault/src/instructions/user/mod.rs
- programs/vault/src/instructions/valuation/mod.rs
- programs/vault/src/instructions/valuation/update_operator_margin_account_position.rs
- programs/vault/src/instructions/valuation/update_operator_wallet_position.rs
- programs/vault/src/instructions/valuation/update_vault_balances.rs
- programs/vault/src/lib.rs
- programs/vault/src/seeds.rs
- programs/vault/src/state/mod.rs
- programs/vault/src/state/operator.rs
- programs/vault/src/state/operator_position.rs
- programs/vault/src/state/pending_withdrawals.rs
- programs/vault/src/state/proposal.rs
- programs/vault/src/state/vault.rs
- programs/vault/src/state/vault_user.rs
- programs/vault/src/utils/margin_accounts.rs
- programs/vault/src/utils/mod.rs
- programs/vault/src/utils/tokens.rs

Out-of-Scope: Changes that are not part of the Pull Request 2437, third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 0299a8a
- fa1b70f
- 1ef1ff7
- fef8113
- 6a40da6
- 4839576
- 295c8a7
- f6d76b1

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

1 1 0 2 6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT FEES HANDLING	CRITICAL	SOLVED - 11/07/2025
USERS MAY PAY PERFORMANCE FEES FROM UNREALIZED PROFITS	HIGH	SOLVED - 11/07/2025
MISSING TOKEN 2022 EXTENSIONS VALIDATION	LOW	PARTIALLY SOLVED - 10/22/2025
INSUFFIENT OPERATOR PERMISSIONS CHECK	LOW	SOLVED - 10/22/2025
INCORRECT PENDING WITHDRAWAL SHARES CALCULATION	INFORMATIONAL	SOLVED - 10/22/2025
RELIANCE ON MANUAL OR OFF-CHAIN ACTIONS	INFORMATIONAL	ACKNOWLEDGED - 11/05/2025
INCORRECT REDUNDANT POSITION FRESHNESS VALIDATION	INFORMATIONAL	SOLVED - 11/23/2025
MISSING INSTRUCTIONS TO WITHDRAW UNCOLLECTED FEES	INFORMATIONAL	FUTURE RELEASE - 10/24/2025
UNNECESSARY TOKEN PROGRAM RESOLUTION	INFORMATIONAL	SOLVED - 11/23/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PASSING UNNECESSARY ACCOUNTS	INFORMATIONAL	SOLVED - 10/22/2025

7. FINDINGS & TECH DETAILS

7.1 INCORRECT FEES HANDLING

// CRITICAL

Description

The vault's management and performance fees accounting is implemented incorrectly, resulting in an **inflated exchange rate** and inaccurate user withdrawals.

`Management fees handling expected scenario`

- A user deposits **1,000 tokens** and receives **1,000 shares** (exchange rate = **1.0**).
- Over one year, a **1% annual management fee** is accrued.
- The total token balance should decrease by 1%, reducing the exchange rate to **0.99**.
- When the user withdraws, they should receive **990 tokens**, reflecting the management fee deduction.

However, in the current implementation, the method `Vault::update_vault` incorrectly **adds** the accrued management fees to the vault's total token balance. Since the vault reserve already contains the uncollected management fees, this addition is **redundant** and causes the vault's total assets to be overstated. This logic error leads to **artificial inflation of the exchange rate**. As a result, users withdrawing funds after management fees are accrued will receive **more tokens than expected**, effectively **bypassing the management fee deduction** and causing accounting inconsistencies.

programs/vault/src/state/vault.rs

```
252 // Deposit tokens are the sum of idle tokens, operator tokens
253 let total_tokens = operator_tokens
254     .checked_add(vault_reserve.amount)
255     .ok_or(crate::ErrorCode::Overflow)?
256     .checked_add(self.uncollected_management_fees)
257     .ok_or(crate::ErrorCode::Overflow)?;
258 self.deposit_tokens = total_tokens;
```

`Performance fees handling expected scenario`

- A user deposits **1,000 tokens** and receives **1,000 shares** (exchange rate = **1.0**).
- The vault's exchange rate later increases to ex. **1.3** due to operator activity and performance gains.
- The user initiates a withdrawal, redeeming nearly all shares (leaving a minimal amount to make sure exchange rate will be re-calculated).
- The withdrawal correctly deducts a **1.5% performance fee** from the user's profit, with this fee remaining in the **vault reserve**.
- The vault reserve therefore holds both the **user's unwithdrawn share tokens** and the **withheld performance fee**.

However, during the next vault update, the method `Vault::update_vault` incorrectly recalculates the total deposited tokens (`vault.deposit_tokens`) as the **sum of**:

- The **vault reserve amount**,
- The **operator's token balance**, and
- **Uncollected management fees**.

Because the vault reserve already includes the **withheld performance fees**, this addition **double-counts** those tokens, artificially inflating the vault's reported total assets and, consequently, the **exchange rate**. This issue leads to an **inaccurate exchange rate**, allowing users to withdraw more tokens than they should be entitled to, effectively reducing the protocol's fee revenue and creating accounting inconsistencies within the vault.

programs/vault/src/state/vault.rs

```
252 | // Deposit tokens are the sum of idle tokens, operator tokens
253 | let total_tokens = operator_tokens
254 |     .checked_add(vault_reserve.amount)
255 |     .ok_or(crate::ErrorCode::Overflow)?
256 |     .checked_add(self.uncollected_management_fees)
257 |     .ok_or(crate::ErrorCode::Overflow)?;
258 | self.deposit_tokens = total_tokens;
```

Proof of Concept

Performance fees flow:

- A user deposits **1,000 tokens** and receives **1,000 shares** (exchange rate = **1.0**).
- The vault's exchange rate later increases to **1.3** due to operator activity and performance gains.
- The user initiates a withdrawal, redeeming nearly all shares (leaving a minimal dust amount).
- The withdrawal correctly deducts a **1.5% performance fee** from the user's profit, with this fee remaining in the **vault reserve**.
- Next vault update incorrectly calculate the deposit tokens and consequently the exchange rate.

```
// Configure vault and the performance fees
sol_vault
    .configure_vault(VaultConfig {
        performance_fee: Some(150), // set performance fees to 1.5 %
        management_fee: None,
        vault_flags: Some(0b00000111),
        deposit_limit: Some(u64::MAX),
        withdrawal_limit: Some(u64::MAX),
        withdrawal_waiting_period: None,
        vault_name: Some(*b"vault                           "),
        oracle: Some(env.sol_oracle),
        minimum_deposit: Some(1_000_000),
        minimum_shares_dust_threshold: Some(1_000),
    })
    .with_signer(&ctx.airspace_authority)
    .send_and_confirm(&ctx.rpc())
    .await?;
// ...

// Deposit user tokens
```

```
let user_deposit = 1000 * LAMPORTS_PER_SOL;
sol_vault.deposit(
    &user_address,
    &user_address,
    None,
    TokenChange::shift(user_deposit),
)
.with_signer(&user)
.send_and_confirm(&ctx.rpc())
.await?;

// ...

// Update operator wallet position to simulate increase in performance
sol_vault
    .update_operator_wallet_position(
        operator_address,
        Number128::from(2000000 * LAMPORTS_PER_SOL),
    )
    .with_signer(&operator_wallet)
    .send_and_confirm(&ctx.rpc())
    .await?;

// ...

// Accrue user performance fees
sol_vault
    .accrue_performance_fees(vault_user_address)
    .without_signer()
    .send_and_confirm(&ctx.rpc())
    .await?;

// ..

// Initiate withdrawal
sol_vault
    .initiate_withdrawal(
        user_address,
        vault.deposit_shares - vault.minimum_shares_dust_threshold - 1, // making sure there is at least
        // vault.deposit_shares, // withdraw all
        None,
    )
    .with_signer(&user)
    .send_and_confirm(&ctx.rpc())
    .await?;

    let reserve_account: anchor_spl::token::TokenAccount =

// ...

// Finalize withdrawal request
sol_vault
    .execute_vault_withdrawal(user_address, 0)
    .with_signer(&user)
    .send_and_confirm(&ctx.rpc())
    .await?;

let reserve_account: anchor_spl::token::TokenAccount =
    get_anchor_account(&ctx.rpc(), &vault.vault_reserve).await?;

let vault: Vault = get_anchor_account(&ctx.rpc(), &sol_vault.address).await?;
let rate = vault.token_to_share_exchange_rate(vault.last_update_timestamp);
println!("==> After user withdrawal");
let reserve_account: anchor_spl::token::TokenAccount =
    get_anchor_account(&ctx.rpc(), &vault.vault_reserve).await?;

println!("reserve_account.amount = {}", reserve_account.amount);
println!("Exchange rate = {}", rate.unwrap());
println!("Deposit tokens = {}", vault.deposit_tokens);
println!("Operator tokens = {}", vault.operator_tokens);
println!("Deposit shares = {}", vault.deposit_shares);
println!(
    "Uncollected perf fees = {}",
```

```
    vault.uncollected_performance_fees
);
println!("-----------");
let user_balance_before = user_token_account.amount;
let user_token_account: anchor_spl::token::TokenAccount =
    get_anchor_account(&ctx.rpc(), &user_sol_ata).await?;
let user_balance_after = user_token_account.amount;
println!(
    "User token account balance after withdrawal = {}",
    user_token_account.amount
);
let withdrawn = user_balance_after - user_balance_before;
println!("withdrawn = {}", withdrawn);
println!("Initial deposit = {user_deposit}");
println!(
    "Profit = Withdrawn - initial deposit = {}",
    user_balance_after - user_deposit
);
println!(
    "Profit in percent = {} %",
    (user_balance_after - user_deposit) as f64 * 100.0 / user_deposit as f64
);
// Update the vault
sol_vault
    .update_vault_balances()
    .without_signer()
    .send_and_confirm(&ctx.rpc())
    .await?;
let reserve_account: anchor_spl::token::TokenAccount =
    get_anchor_account(&ctx.rpc(), &vault.vault_reserve).await?;

let vault: Vault = get_anchor_account(&ctx.rpc(), &sol_vault.address).await?;
let rate = vault.token_to_share_exchange_rate(vault.last_update_timestamp);
println!("==> After final vault update:");
let reserve_account: anchor_spl::token::TokenAccount =
    get_anchor_account(&ctx.rpc(), &vault.vault_reserve).await?;

println!("reserve_account.amount = {}", reserve_account.amount);
println!("Exchange rate = {}", rate.unwrap());
println!("Deposit tokens = {}", vault.deposit_tokens);
println!("Operator tokens = {}", vault.operator_tokens);
println!("Deposit shares = {}", vault.deposit_shares);
println!(
    "Uncollected perf fees = {}",
    vault.uncollected_performance_fees
);
println!("-----------");
```

```
==> After user withdrawal
reserve_account.amount = 4497581942
Exchange rate = 1.3006993006
Deposit tokens = 1302
Operator tokens = 0
Deposit shares = 1001
Uncollected perf fees = 4500000000
-----------
User token account balance after withdrawal = 1295502418058
withdrawn = 1295502418058
Initial deposit = 1000000000000
Profit = Withdrawn - initial deposit = 295502418058
Profit in percent = 29.5502418058 %
==> After final vault update:
reserve_account.amount = 4497581942
Exchange rate = 4493088.8531468531
Deposit tokens = 4497581942
Operator tokens = 0
Deposit shares = 1001
Uncollected perf fees = 4500000000
-----------
```

Management fees flow:

1. A user deposits **1,000 tokens** and receives **1,000 shares** (exchange rate = **1.0**).
2. Over one year, a **1% annual management fee** is accrued.
3. The total token balance should decrease by 1%, reducing the exchange rate to **0.99**.
4. However the deposit tokens value increases and the exchange rate remains 1.0.

```rust
// Configure the vault and the management fee
sol_vault
    .configure_vault(VaultConfig {
        performance_fee: None,
        management_fee: Some(150), // set management fee
        vault_flags: Some(0b00000111),
        deposit_limit: Some(u64::MAX),
        withdrawal_limit: Some(u64::MAX),
        withdrawal_waiting_period: None,
        vault_name: Some(*b"vault                           "),
        oracle: Some(env.sol_oracle),
        minimum_deposit: Some(1_000_000),
        minimum_shares_dust_threshold: Some(1_000),
    })
    .with_signer(&ctx.airspace_authority)
    .send_and_confirm(&ctx.rpc())
    .await?;

// ...
```

```rust
// Deposit user tokens
let user_deposit = 1000 * LAMPORTS_PER_SOL;
sol_vault.deposit(
    &user_address,
    &user_address,
    None,
    TokenChange::shift(user_deposit),
)
.with_signer(&user)
.send_and_confirm(&ctx.rpc())
.await?;

let vault: Vault = get_anchor_account(&ctx.rpc(), &sol_vault.address).await?;
let rate = vault.token_to_share_exchange_rate(vault.last_update_timestamp);
println!("==> Before time warp");
println!("Exchange rate = {}", rate.unwrap());
println!("Deposit tokens = {}", vault.deposit_tokens);
println!("Operator tokens = {}", vault.operator_tokens);
println!("Deposit shares = {}", vault.deposit_shares);
println!(
    "Uncollected management fees = {}",
    vault.uncollected_management_fees
);
println!("-----------");

// Jump forward 1 year in time and update the vault to simulate 1 year of management fees accrual
let clock = ctx.rpc().get_clock().await?;

ctx.solana
    .context_mut()
    .await
    .warp_to_timestamp(clock.unix_timestamp + 3600 * 24 * 365)
    .await?;

sol_vault
    .update_vault_balances()
    .without_signer()
    .send_and_confirm(&ctx.rpc())
    .await?;

let vault: Vault = get_anchor_account(&ctx.rpc(), &sol_vault.address).await?;
let rate = vault.token_to_share_exchange_rate(vault.last_update_timestamp);
println!("==> After time warp");
println!("Exchange rate = {}", rate.unwrap());
println!("Deposit tokens = {}", vault.deposit_tokens);
println!("Operator tokens = {}", vault.operator_tokens);
println!("Deposit shares = {}", vault.deposit_shares);
println!(
    "Uncollected management fees = {}",
    vault.uncollected_management_fees
);
println!("-----------");
```

```
==> Before time warp
Exchange rate = 1.0
Deposit tokens = 1000000000000
Operator tokens = 500000000000
Deposit shares = 1000000000000
Uncollected management fees = 0
-----------
* accrued user performance fees = 0
==> After time warp
Exchange rate = 1.0
Deposit tokens = 1015000000000
Operator tokens = 500000000000
Deposit shares = 1000000000000
Uncollected management fees = 15000000000
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:H (9.4)

Recommendation

To address this finding, it is recommended to update the vault's **fee accounting logic** to ensure that both **management** and **performance fees** are handled correctly. Accrued management and withheld performance fees should **not be added to the vault's total token balance**, as they are already represented within the vault reserve. The vault's total asset value should reflect only **active user deposits and operator balances**, ensuring the **exchange rate accurately accounts for fee deductions** and maintains **consistent and transparent vault accounting**.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by correcting the management and performance fees accounting and ensuring the program correctly tracks the uncollected fee amounts and excludes these amounts during exchange rate calculation.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/0299a8ad54e11eb70c1536cba3f4b38d065c6153

7.2 USERS MAY PAY PERFORMANCE FEES FROM UNREALIZED PROFITS

// HIGH

Description

The `execute_vault_withdrawal` instruction allows users to finalize their previously initiated withdrawal requests. During this process, the instruction updates performance fees and deducts them from the total token amount to be withdrawn.

The permissionless instruction `accrue_performance_fees` can also be called at any time before the withdrawal to calculate performance fees based on the **current exchange rate**. However, performance fees are only updated when the current exchange rate exceeds the previously recorded **"high-water mark"** (the last highest exchange rate).

This creates a potential vulnerability: if performance fees are accrued while the exchange rate is high, but the user later withdraws after the exchange rate drops, the user will still pay fees on **unrealized profits**. Consequently, users may incur excessive performance fees or be charged fees despite experiencing a net loss at the time of withdrawal.

```
168 // Collect the performance fees proportionately to the remaining balance
169 let vault_user = &mut ctx.accounts.vault_user;
170 let performance_fees_to_withhold = if shares == vault_user.total_shares {
171     vault_user.accrued_performance_fees
172 } else if vault_user.accrued_performance_fees > 0 {
173     let withdrawal_tokens = Number128::from_decimal(gross_withdrawal_tokens, 0);
174     let fees = Number128::from_decimal(vault_user.accrued_performance_fees, 0);
175     let total_tokens = Number128::from_decimal(total_tokens, 0);
176     withdrawal_tokens
177         .safe_div(total_tokens)?
178         .safe_mul(fees)?
179         .as_u64(0)
180         .min(vault_user.accrued_performance_fees)
181 } else {
182     0
183 };
184
185 // Shares were burned, so remove them from the user's share tally.
186 vault_user.total_shares = vault_user
187     .total_shares
188     .checked_sub(shares)
189     .ok_or(crate::ErrorCode::Overflow)?;
190
191 let net_withdrawal_tokens =
192     gross_withdrawal_tokens.saturating_sub(performance_fees_to_withhold);
193 vault_user.accrued_performance_fees = vault_user
194     .accrued_performance_fees
195     .saturating_sub(performance_fees_to_withhold);
196 msg!(
197     "Gross tokens {}, fees held {}",
198     gross_withdrawal_tokens,
199     performance_fees_to_withhold
200 );
201
202 tokens::transfer_tokens(
203     ctx.accounts.underlying_mint_token_program.to_account_info(),
```

```
    ctx.accounts.underlying_mint.to_account_info(),
    ctx.accounts.vault_reserve.to_account_info(),
    ctx.accounts
        .destination_underlying_token_account
        .to_account_info(),
    ctx.accounts.vault.to_account_info(),
    ctx.accounts.underlying_mint.decimals,
    net_withdrawal_tokens,
    Some(&seeds),
)?;
```

```
pub fn accrue_user_performance_fees(
    &mut self,
    vault_user: &mut VaultUser,
    timestamp: i64,
) -> Result<()> {
    // Get the rate
    let vault_exchange_rate = self.token_to_share_exchange_rate(timestamp)?;

    // Compare the rate with the user's stored rate.
    if vault_exchange_rate > *vault_user.last_performance_fee_rate() {
        // Accrue the fee
        // = shares * (new_rate - old_rate) * performance fee
        let rate_delta =
            vault_exchange_rate.safe_sub(*vault_user.last_performance_fee_rate())?;
        let shares = Number128::from_decimal(vault_user.total_shares, 0);
        let performance_fee = Number128::from_bps(self.performance_fee);
        let performance_fee = shares.safe_mul(rate_delta)?.safe_mul(performance_fee)?;
        let performance_fee_tokens = performance_fee.as_u64(0);

        vault_user.accrued_performance_fees += performance_fee_tokens;
        *vault_user.last_performance_fee_rate_mut() = vault_exchange_rate;

        // Increment the vault's performance fees.
        self.uncollected_performance_fees = self
            .uncollected_performance_fees
            .checked_add(performance_fee_tokens)
            .ok_or(crate::ErrorCode::Overflow)?;
    }

    vault_user.last_update_timestamp = timestamp;

    Ok(())
}
```

Proof of Concept

1. User deposits to vault at share exchange rate 1.0.
2. Share exchange rate increases.
3. User performance fees are accrued.
4. Share exchange rate decreases.
5. User withdraws.

```
// ...
// Create and configure a vault
    sol_vault
        .create_vault(airspace_authority)
        .with_signer(&ctx.airspace_authority)
        .send_and_confirm(&ctx.rpc())
        .await?;
    sol_vault
        .configure_vault(VaultConfig {
```

```
            performance_fee: Some(150), // set a performance fee
            management_fee: None,
            vault_flags: Some(0b00000111),
            deposit_limit: Some(u64::MAX),
            withdrawal_limit: Some(u64::MAX),
            withdrawal_waiting_period: None, // deactivate waiting period
            vault_name: Some(*b"vault                           "),
            oracle: Some(env.sol_oracle),
            minimum_deposit: Some(1_000_000),
            minimum_shares_dust_threshold: Some(1_000),
        })
        .with_signer(&ctx.airspace_authority)
        .send_and_confirm(&ctx.rpc())
        .await?;
// ...
// User deposits to the vault
    let user_deposit = 1000 * LAMPORTS_PER_SOL;
    vec![
       sol_vault.deposit(
            &user_address,
            &user_address,
            None,
            TokenChange::shift(user_deposit),
        ),
    ]
    .with_signer(&user)
    .send_and_confirm(&ctx.rpc())
    .await?;
// ...
// Update operator's wallet position to simulate increase of the share exchange rate
    sol_vault
        .update_operator_wallet_position(
            operator_address,
            Number128::from(2000000 * LAMPORTS_PER_SOL),
        )
        .with_signer(&operator_wallet)
        .send_and_confirm(&ctx.rpc())
        .await?;
// ...
// Accrue user performance fee at high share exchange rate
sol_vault
        .accrue_performance_fees(vault_user_address)
        .without_signer()
        .send_and_confirm(&ctx.rpc())
        .await?;
// ...
// Update operator's wallet position to simulate decrease of the share exchange rate
sol_vault
        .update_operator_wallet_position(
            operator_address,
            Number128::from(1500000 * LAMPORTS_PER_SOL),
        )
        .with_signer(&operator_wallet)
        .send_and_confirm(&ctx.rpc())
        .await?;
// ...
// Initiate withdrawal of all shares
    let pending_pda = sol_vault.derive_pending_withdrawals(&user_address);
    sol_vault
        .create_vault_pending_withdrawal(user_address, user_address)
        .with_signer(&user)
        .send_and_confirm(&ctx.rpc())
        .await?;

    sol_vault
        .initiate_withdrawal(
            user_address,
            vault.deposit_shares, // withdraw all
            None,
        )
        .with_signer(&user)
        .send_and_confirm(&ctx.rpc())
        .await?;
```

```
// ...
// Finalize the withdrawal request
    let vault: Vault = get_anchor_account(&ctx.rpc(), &sol_vault.address).await?;
    let rate = vault.token_to_share_exchange_rate(vault.last_update_timestamp)?;

    let gross_tokens = Number128::from(vault_user.total_shares) * rate;
    let profit_before_fee = gross_tokens - Number128::from(user_deposit);
    let perf_fee = profit_before_fee * Number128::from_bps(vault.performance_fee);
    let net_tokens = gross_tokens - perf_fee;

    sol_vault
        .execute_vault_withdrawal(user_address, 0)
        .with_signer(&user)
        .send_and_confirm(&ctx.rpc())
        .await?;
// ...
// Perform checks
    let vault: Vault = get_anchor_account(&ctx.rpc(), &sol_vault.address).await?;
    let rate = vault.token_to_share_exchange_rate(vault.last_update_timestamp);

    let user_balance_before = user_token_account.amount;
    let user_token_account: anchor_spl::token::TokenAccount =
        get_anchor_account(&ctx.rpc(), &user_sol_ata).await?;
    let user_balance_after = user_token_account.amount;
    let withdrawn = user_balance_after - user_balance_before;

    assert_eq!(net_tokens, Number128::from(withdrawn));
```

```
[2025-10-15T11:07:29.647461000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: ExecuteVaultWithdrawal
[2025-10-15T11:07:29.651084000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb invoke [2]
[2025-10-15T11:07:29.651401000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: Burn
[2025-10-15T11:07:29.652256000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb consumed 5324 of 170439 compute units
[2025-10-15T11:07:29.652279000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb success
[2025-10-15T11:07:29.652485000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Gross tokens 1050000000000, fees held 4500000000
[2025-10-15T11:07:29.652973000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [2]
[2025-10-15T11:07:29.653371000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: TransferChecked
[2025-10-15T11:07:29.654386000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 6239 of 161154 compute units
[2025-10-15T11:07:29.654410000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA success
[2025-10-15T11:07:29.655109000Z DEBUG solana_runtime::message_processor::stable_log] Program gwv1ybUe2JVEpjdWARK1PjZUVY5xdNUCRhu24tgYtxa consumed 48470 of 200000 compute units
[2025-10-15T11:07:29.655133000Z DEBUG solana_runtime::message_processor::stable_log] Program gwv1ybUe2JVEpjdWARK1PjZUVY5xdNUCRhu24tgYtxa success
thread 'test_performance_fees' panicked at tests/hosted/tests/halborn.rs:847:5:
assertion `left == right` failed
  left: 104.925
 right: 104.55
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   0.759s] 1 test run: 0 passed, 1 failed, 259 skipped
        FAIL [   0.748s] hosted-tests::halborn test_performance_fees
error: test run failed
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:M/Y:M (7.5)

Recommendation

To address this issue, allow the performance fee to be calculated based on the exchange rate at the time of withdrawal. This will charge a fair performance fee and prevent charging a performance fee on unrealized profit.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by implementing a time-locking mechanism where the performance fees can be accrued only once per 28 days. This limits intentional performance fee accrual

abuse when the exchange rate is temporarily high but at the same time charges performance fees on dormant accounts periodically.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/fa1b70f028282bf054a21a63e3e2ef7ad79b9ccb

7.3 MISSING TOKEN 2022 EXTENSIONS VALIDATION

// LOW

Description

The `create_vault` instruction allows an authorized user to create a new vault and specify any mint as the underlying asset. However, the instruction does **not validate which Token-2022 extensions** are enabled on the selected mint. This omission allows the use of tokens with **potentially dangerous extensions**, which could compromise the security or integrity of the protocol.

Using an underlying asset with certain Token-2022 extensions could introduce significant risks:

- **PermanentDelegate** – Allows a delegate to transfer or burn assets locked in the vault, potentially leading to loss of funds.
- **Pausable** – Enables a mint authority to pause token transfers, which could disrupt normal protocol operations.
- **TransferFee** – Applies fees to transfers, potentially causing **accounting inconsistencies** or **unexpected behavior** when assets move in or out of the vault.
- **TransferHook** – May require additional accounts needed by the transfer hook program, potentially causing a **denial of service.**

```
68 | /// Underlying asset mint
69 | pub underlying_mint: Box<InterfaceAccount<'info, Mint>>,
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:C/D:C/Y:N (3.0)

Recommendation

To address this finding, it is recommended to validate the Token 2022 extensions during vault creation and disallow adding mints with potentially dangerous and unsupported extensions. In case underlying mints with potentially dangerous extensions are required (such as the PYUSD with the PermanentDelegate extension), implement a whitelist to make sure only trusted mints can be used.

Remediation Comment

PARTIALLY SOLVED: The **Glow team** partially solved this finding by adding validation of the **TransferFee** Token extension. However, the following extensions are still not fully validated, and their validation is planned for future releases:

- **PermanentDelegate**

- **Pausable**
- **TransferHook**

Because this instruction can only be invoked by an authorized account, and the effects of these token extensions are well understood, the lack of full validation is considered a **low-risk** issue.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/1ef1ff76a3c8395fa3b14fdb1abd6e2bbb7a53c6

7.4 INSUFFIENT OPERATOR PERMISSIONS CHECK

// LOW

Description

The program requires that a vault operator hold the **OPERATE_VAULTS** permission. This permission is correctly verified when assigning an operator admin to a vault.
However, it is **not validated** in several other operator instructions — most importantly:

- `update_operator_wallet_position`
- `operator_withdraw_from_vault`
- `operator_transfer_to_margin`

These instructions allow an operator to **transfer tokens from the vault** and **manipulate the valuation of operator positions**.

Failing to verify the required **OPERATE_VAULTS** permission in these cases could allow an operator whose permissions were revoked to **continue performing privileged actions**, such as **withdrawing funds** or **altering the vault's share exchange rate**, leading to potential malicious activity.

programs/vault/src/instructions/valuation/update_operator_wallet_position.rs

```
39 | /// The operator whose position is being updated
40 | #[account(
41 |     mut,
42 |     seeds = [VAULT_OPERATOR_SEED, vault.key().as_ref()],
43 |     bump,
44 |     has_one = vault,
45 | )]
46 | pub operator: AccountLoader<'info, VaultOperator>,
```

BVSS

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

Recommendation

To address this issue, it is recommended to validate that an operator has valid permissions when invoking the instructions `update_operator_wallet_position`, `operator_withdraw_from_vault` and `operator_transfer_to_margin` and consider restricting other operator-related instructions.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by explicitly validating that an operator has OPERATE_VAULTS permission before executing the `update_operator_wallet_position`, `operator_withdraw_from_vault` or `operator_transfer_to_margin` instruction.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/fef8113b0dca5e19f11c6c9e9f3a82f05f8030a1

7.5 INCORRECT PENDING WITHDRAWAL SHARES CALCULATION

// INFORMATIONAL

Description

The `cancel_vault_pending_withdrawal` instruction allows a user to cancel a previously initialized withdrawal request. However, it incorrectly logs the total number of pending withdrawal shares. When a withdrawal is canceled, the refunded shares are **added** to the total instead of being **subtracted**, resulting in an inaccurate log value.

The severity of this issue is **informational**, since the `vault_user.pending_withdrawal_shares` value is **not used in any on-chain calculations**. However, this inconsistency may affect **off-chain consumers** of the data—such as front-end applications—potentially leading to **display errors or unexpected behavior**.

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

```
111 let refund_shares = pending.pending_shares;
112 let vault_user = &mut ctx.accounts.vault_user;
113 vault_user.pending_withdrawal_shares = vault_user
114     .pending_withdrawal_shares
115     .checked_add(refund_shares)
116     .ok_or(crate::ErrorCode::Overflow)?;
```

BVSS

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

Recommendation

To address this finding, it is recommended to decrease the `vault_user.pending_withdrawal_shares` value by the number of shares to refund.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by decreasing the `vault_user.pending_withdrawal_shares` value by the number of shares to refund.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/6a40da622c2aa62f095c3aa1fa88de061807529a

7.6 RELIANCE ON MANUAL OR OFF-CHAIN ACTIONS

// INFORMATIONAL

Description

At its current development stage, the protocol relies on **manual and off-chain interventions**. These include the following:

- A **permissioned operator** is required to withdraw user-deposited vault funds in order to perform external fund management operations outside the Glow protocol.
- A **permissioned operator** is also responsible for providing accurate valuations of their wallet positions.

As a result, users must place **full trust in the protocol's operator** to supply correct valuation data and manage funds responsibly.

BVSS

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

Recommendation

To address this finding, it is recommended to implement measures that **minimize or completely eliminate the need for manual interventions**, and to **provide clear, publicly available documentation** describing the protocol's functionality.

Remediation Comment

ACKNOWLEDGED The **Glow team** acknowledged this finding.

7.7 INCORRECT REDUNDANT POSITION FRESHNESS VALIDATION

// INFORMATIONAL

Description

During deposit or withdrawal interactions with the vault, the program verifies that the operator's position valuations are up to date and contain fresh data. However, the helper methods `VaultOperator::increase_position_tokens` and `VaultOperator::decrease_position_tokens` perform this check incorrectly.

These methods validate that the position was updated **within the next 30 seconds**, rather than **within the last 30 seconds**, causing the freshness check to **always pass**, even when the position data is stale.

The severity of this issue is **informational**, since a correct freshness validation is also performed by the `Vault::update_vault` method, which is called immediately after `increase_position_tokens` and `decrease_position_tokens`. As a result, this logic error does not impact the program's behavior but may cause confusion or reduce code reliability.

programs/vault/src/state/operator.rs

```
158 pub fn increase_position_tokens(
159     &mut self,
160     destination_kind: PositionDestinationKind,
161     destination_address: Pubkey,
162     amount: u64,
163     timestamp: i64,
164 ) -> Result<()> {
165     let position = self.get_position_mut(destination_kind, destination_address)?;
166     // The position must have been updated in the last 30 seconds
167     require!(
168         timestamp + 30 >= position.last_update_ts,
169         ErrorCode::PositionStale
170     );
```

programs/vault/src/state/operator.rs

```
181 pub fn decrease_position_tokens(
182     &mut self,
183     destination_kind: PositionDestinationKind,
184     destination_address: Pubkey,
185     amount: u64,
186     timestamp: i64,
187 ) -> Result<()> {
188     let position = self.get_position_mut(destination_kind, destination_address)?;
189     // The position must have been updated in the last 30 seconds
190     require!(
191         timestamp + 30 >= position.last_update_ts,
192         crate::ErrorCode::PositionStale
193     );
```

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:L/D:N/Y:N (0.6)

Recommendation

To address this finding, it is recommended to either correct and align the freshness check in the `increase_position_tokens` and `decrease_position_tokens` methods to the check implemented in `Vault::update_vault` method or alternatively remove the check from the `increase_position_tokens` and `decrease_position_tokens` methods and keep it only in the `Vault::update_vault` method.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by correcting the freshness check in the `increase_position_tokens` and `decrease_position_tokens` methods.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/4839576eb51eff693a244440af672a4f1e19cabd

7.8 MISSING INSTRUCTIONS TO WITHDRAW UNCOLLECTED FEES

// INFORMATIONAL

Description

The protocol can collect **fund management** and **performance fees** depending on each vault's configuration. These fees are recorded and stored in the vault's **reserve account**.

However, the program does **not** include any instruction that allows these accumulated fees to be withdrawn. As a result, the fees become effectively **locked**, since the reserve account is owned by a **Program-Derived Address (PDA)**, and any transfer from it must be **authorized by the program itself**.

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:C/I:N/D:N/Y:N (0.5)

Recommendation

To address this finding, it is recommended to implement dedicated instructions to withdraw the management and performance fees and update the vault state.

Remediation Comment

FUTURE RELEASE: The **Glow team** acknowledged this finding and plans to remediate this finding in the next release.

7.9 UNNECESSARY TOKEN PROGRAM RESOLUTION

// INFORMATIONAL

Description

Several instructions use the `tokens::resolve_token_program` helper function to determine the correct Token program for a given mint. However, this **on-chain program resolution is unnecessary**, since the same logic can be performed **off-chain** by simply providing the appropriate Token program (`Token` or `Token2022`) directly to the instruction. During token related operations, the token programs already verify that the correct token program is used and return an error if not.

Performing this check on-chain **increases compute unit usage** and therefore **raises transaction costs** without providing any additional security or functional benefit.

programs/vault/src/instructions/operator/operator_withdraw_from_vault.rs

```
86 | // Token programs (classic + 2022)
87 | pub token_program: Interface<'info, TokenInterface>,
88 | pub token_2022_program: Interface<'info, TokenInterface>,
```

```
133 | // Resolve correct token program (Token vs Token-2022) by the mint owner
134 | let underlying_mint = ctx.accounts.underlying_mint.to_account_info();
135 | let token_program = tokens::resolve_token_program(
136 |     &underlying_mint,
137 |     &ctx.accounts.token_program.to_account_info(),
138 |     &ctx.accounts.token_2022_program.to_account_info(),
139 | )?;
```

BVSS

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

Recommendation

To address this finding, it is recommended to follow these guidelines:

- If the required Token program is known in advance, require the given program by using the appropriate Anchor data types `Program<'info, Token>` or `Program<'info, Token2022>` .
- If the required Token program is not known in advance, use the `Interface<'info, TokenInterface>` Anchor data type.
- If an instruction may require both `Token` and `Token2022` programs, but it is not known in advance, which mint will require which program, use separate `Interface` account types for each mint such as and omit the `token::resolve_token_program` helper function.

```
pub mint1_token_program: Interface<'info, TokenInterface>,
pub mint2_token_program: Interface<'info, TokenInterface>,
```

Remediation Comment

SOLVED: The **Glow team** resolved this finding by removing the `tokens::resolve_token_program` helper function and requiring the corresponding token program to be passed.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/295c8a7a4729cd276e98acb0333a7bfc52b9178c

7.10 PASSING UNNECESSARY ACCOUNTS

// INFORMATIONAL

Description

The instructions `initiate_withdrawal` and `cancel_pending_withdrawal` require the `system_program` account that is not used and thus is unnecessary. Also, both instruction require the `underlying_mint` account. However this account is used only for seeds derivation and is not read or written to. It is therefore more efficient to pass this account as instruction parameter instead as an instruction account.

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

```
85 | /// Underlying mint (asset) from the vault
86 | pub underlying_mint: UncheckedAccount<'info>, // equals vault.underlying_mint
87 |
88 | pub token_program: Interface<'info, TokenInterface>,
89 | pub system_program: Program<'info, System>
```

BVSS

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

Recommendation

To address this finding, it is recommended to remove any unused accounts. In case only public key is needed, it is preferable to pass it as instruction parameter.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by removing the unnecessary accounts.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/f6d76b1f42486c14161da92bc318b1e7a479ff70

8. AUTOMATED TESTING

Static Analysis Report

Description

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

Cargo Audit Results

ID	CRATE	DESCRIPTION
RUSTSEC-2025-0024	crossbeam-channel	crossbeam-channel: double free on Drop
RUSTSEC-2024-0344	curve25519-dalek	Timing variability in `curve25519-dalek`'s `Scalar29::sub` / `Scalar52::sub`
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
RUSTSEC-2025-0022	openssl	Use-After-Free in `Md::fetch` and `Cipher::fetch`
RUSTSEC-2025-0009	ring	Some AES functions may panic when overflow checking is enabled.
RUSTSEC-2025-0009	ring	Some AES functions may panic when overflow checking is enabled.
RUSTSEC-2025-0055	tracing-subscriber	Logging user input may result in poisoning logs with ANSI escape sequences

---

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