// Security Assessment

01.16.2026 - 01.21.2026

LRT Migration Review & Fee Calculation

Blueprint Finance

HALBORN

LRT Migration Review & Fee Calculation - Blueprint Finance

Prepared by: HALBORN

Last Updated 02/11/2026

Date of Engagement: January 16th, 2026 - January 21st, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	2	0	3

TABLE OF CONTENTS

1. Introduction

`Blueprint` engaged Halborn to conduct a security assessment on their `glow-lrt` program beginning on January 16th, 2026 and ending on January 21st, 2026. The security assessment was scoped to the smart contracts provided in the GitHub repository glow-v1, commit hash, and further details can be found in the Scope section of this report.

The `Blueprint team` is releasing a new version of their `glow-lrt` Solana program. This program implements a Liquid Restaking Token (LRT) system that allows users to deposit SOL and receive shares representing their stake, with margin functionality enabling deposits and withdrawals through margin accounts while tracking performance fees based on profits.

2. Assessment Summary

Halborn was provided 3 business days 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 Program.
- Ensure that smart contract functionality operates as intended.

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

- `Add pool identifier to MarginLrtState PDA seeds to prevent cross-pool state collision.`
- `Use proportional original deposit amount instead of current value when decrementing deposited_sol in margin withdrawals.`

3. Test Approach And Methodology

Halborn performed a combination of manual review and security testing based on scripts to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Differences analysis using GitLens to have a proper view of the differences between the mentioned commits
- Graphing out functionality and programs logic/connectivity/functions along with state changes

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: 814eda8

(c) Items in scope:

- programs/lrt/src/errors.rs
- programs/lrt/src/events.rs
- programs/lrt/src/instructions/admin/accept_account_change.rs
- programs/lrt/src/instructions/admin/configure_pool.rs
- programs/lrt/src/instructions/admin/freeze_tokens.rs
- programs/lrt/src/instructions/admin/initialize.rs
- programs/lrt/src/instructions/admin/initialize_pool_mints.rs
- programs/lrt/src/instructions/admin/propose_account_change.rs
- programs/lrt/src/instructions/admin/thaw_tokens.rs
- programs/lrt/src/instructions/admin/transfer_to_treasury.rs
- programs/lrt/src/instructions/admin_staking/solayer_restake.rs
- programs/lrt/src/instructions/admin_staking/solayer_unrestake.rs
- programs/lrt/src/instructions/margin/accept_margin_fee_receiver_change.rs
- programs/lrt/src/instructions/margin/configure_margin_pool_authority.rs
- programs/lrt/src/instructions/margin/create_margin_lrt_state.rs
- programs/lrt/src/instructions/margin/initialize_margin.rs
- programs/lrt/src/instructions/margin/margin_cancel_pending_withdrawal.rs
- programs/lrt/src/instructions/margin/margin_deposit.rs
- programs/lrt/src/instructions/margin/margin_execute_withdraw.rs
- programs/lrt/src/instructions/margin/margin_init_withdraw.rs
- programs/lrt/src/instructions/margin/margin_instant_withdraw.rs
- programs/lrt/src/instructions/margin/margin_refresh_lrt_position.rs
- programs/lrt/src/instructions/margin/margin_set_deposit_limit.rs
- programs/lrt/src/instructions/margin/propose_margin_fee_receiver_change.rs
- programs/lrt/src/instructions/migrations/migrate_lrt_authority.rs
- programs/lrt/src/instructions/migrations/migrate_lrt_pool.rs
- programs/lrt/src/instructions/migrations/migrate_oracle.rs
- programs/lrt/src/instructions/migrations/migrate_withdrawal.rs
- programs/lrt/src/instructions/oracle/create_oracle.rs
- programs/lrt/src/instructions/oracle/update_oracle.rs
- programs/lrt/src/instructions/user/cancel_pending_withdrawal.rs
- programs/lrt/src/instructions/user/close_pending_withdrawal.rs
- programs/lrt/src/instructions/user/create_pending_withdrawals.rs
- programs/lrt/src/instructions/user/deposit_sol.rs
- programs/lrt/src/instructions/user/deposit_stake.rs
- programs/lrt/src/instructions/user/execute_withdrawal.rs

- programs/lrt/src/instructions/user/initiate_withdrawal.rs
- programs/lrt/src/instructions/user/instant_withdrawal.rs
- programs/lrt/src/lib.rs
- programs/lrt/src/seeds.rs
- programs/lrt/src/state.rs
- programs/lrt/src/state/pending_withdrawals.rs
- programs/lrt/src/utils/tests.rs

REMEDIATION COMMIT ID:

- bc47db4
- 921d07b
- 0b99014
- 17224ae
- 80cab93

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	2	0

INFORMATIONAL
3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING POOL IDENTIFIER IN MARGINLRTSTATE PDA SEEDS CAUSES CROSS-POOL STATE COLLISION	MEDIUM	SOLVED - 02/05/2026
SHARE PRICE APPRECIATION CAUSES UNDERFLOW IN DEPOSITED SOL TRACKING, BLOCKING LEGITIMATE WITHDRAWALS	MEDIUM	SOLVED - 02/05/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
WRONG ACCOUNT USED IN RENT BALANCE CHECK CAUSES MIGRATION FAILURES	INFORMATIONAL	SOLVED - 02/05/2026
MISSING VALIDATION BETWEEN PROPOSER ACCOUNT AND STORED PROPOSER FIELD	INFORMATIONAL	SOLVED - 02/05/2026
MISSING MARGINACCOUNT TYPE VALIDATION IN CREATE_MARGIN_LRT_STATE	INFORMATIONAL	SOLVED - 02/05/2026

7. FINDINGS & TECH DETAILS

7.1 MISSING POOL IDENTIFIER IN MARGINLRTSTATE PDA SEEDS CAUSES CROSS-POOL STATE COLLISION

// MEDIUM

Description

The `create_margin_lrt_state` instruction creates a `MarginLrtState` account to track deposit information for performance fee calculations.

As shown in the code snippet below, the PDA is derived using only `MARGIN_LRT_STATE_SEED` and `margin_account` as seeds, without including a pool identifier.

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

Copy Code

```
30      #[account(
31          init,
32          payer = payer,
33          space = 8 + std::mem::size_of::<MarginLrtState>(),
34          seeds = [
35              MARGIN_LRT_STATE_SEED,
36              // TODO: add the pool authority to the seeds
37              margin_account.key().as_ref(),
38          ],
39          bump
40      )]
41      pub margin_lrt_state: Box<Account<'info, MarginLrtState>>,
```

The `MarginLrtState` tracks `deposited_sol` and a `deposits` array for FIFO debt calculation.

If the protocol supports multiple LRT pools in the future, a margin account would share the same `MarginLrtState` across all pools since no pool-specific identifier is included in the PDA derivation.

If this change is implemented after users have already created `MarginLrtState` accounts, a migration process would be required to:

1. Create new `MarginLrtState` accounts with the updated PDA derivation (including pool identifier)
2. Migrate existing deposit tracking data to the new accounts
3. Close the old accounts and refund rent

This migration adds complexity and risk.

Therefore, it is recommended to implement this fix before deploying any additional LRT pools, ideally while the protocol still operates with a single pool to avoid migration overhead.

If the protocol expands to support multiple LRT pools and a user interacts with different pools using the same margin account, the deposit tracking would be corrupted:

1. The `deposited_sol` field would aggregate deposits from all pools, causing incorrect performance fee calculations since profit is calculated as `current_value - deposited_sol - debt`.
2. The `deposits` array would mix deposit records from different pools with different `borrowed_rate` values, causing incorrect FIFO debt calculations.
3. Withdrawing from one pool would incorrectly affect the deposit tracking state of another pool.

Proof of Concept

Intended PoC

This is a prototype POC that demonstrates the expected bug behavior.

A full implementation would require creating two complete LRT pool environments (oracles, fee receivers, margin authorities, etc.), which is beyond the scope of the current test framework designed by the `Blueprint Finance` for single-pool testing.

The prototype shows that the same `MarginLrtState` PDA would be derived for any pool when using the same `margin_account`, confirming the cross-pool state collision issue.

PoC Code

Copy Code

```
// Create MarginLrtState (only once - shared across pools!)
user.lrt_create_margin_lrt_state(&pool_a.pool).await?;

// Deposit 10 SOL into Pool A
let deposit_amount_a = 10 * LAMPORTS_PER_SOL;
user.lrt_deposit(&pool_a.pool, deposit_amount_a, fee_receiver_a).await?;

let state_after_a = get_margin_lrt_state(&ctx, margin_account).await?;
assert_eq!(state_after_a.deposited_sol, 10 * LAMPORTS_PER_SOL);
// deposited_sol = 10 SOL ✓

// Deposit 5 SOL into Pool B (uses SAME MarginLrtState!)
let deposit_amount_b = 5 * LAMPORTS_PER_SOL;
user.lrt_deposit(&pool_b.pool, deposit_amount_b, fee_receiver_b).await?;

let state_after_b = get_margin_lrt_state(&ctx, margin_account).await?;
// BUG: deposited_sol = 15 SOL (10 + 5, mixed from both pools!)
assert_eq!(state_after_b.deposited_sol, 15 * LAMPORTS_PER_SOL);
```

BVSS

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

Recommendation

Before deploying additional LRT pools, it is recommended to add a pool identifier (such as `lrt_margin_authority` or `pool`) to the PDA seeds:

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

Copy Code

```
30      #[account(
31          init,
32          payer = payer,
33          space = 8 + std::mem::size_of::<MarginLrtState>(),
34          seeds = [
35              MARGIN_LRT_STATE_SEED,
36              lrt_margin_authority.key().as_ref(),
37              margin_account.key().as_ref(),
38          ],
39          bump
40      )]
41      pub margin_lrt_state: Box<Account<'info, MarginLrtState>>,
```

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/bc47db4da2b8fca53ca1963f952a8f5fc7c243cf

7.2 SHARE PRICE APPRECIATION CAUSES UNDERFLOW IN DEPOSITED SOL TRACKING, BLOCKING LEGITIMATE WITHDRAWALS

// MEDIUM

Description

The Glow LRT program provides margin functionality that allows users to deposit SOL into an LRT (Liquid Restaking Token) pool through their margin accounts.

The system tracks the original deposited amount in a `deposited_sol` field within the `MarginLrtState` account to calculate performance fees based on profits.

When users withdraw from the pool via `margin_instant_withdraw` or `margin_init_withdraw`, the handlers decrement `deposited_sol` by calling `shares_to_assets(shares)`, which converts shares to their current SOL value based on the oracle exchange rate.

However, this calculation returns the current value of the shares, not the proportional original deposit amount.

As shown in the code snippets below, when the share price appreciates (e.g., due to staking rewards accumulation), the current value of shares exceeds the original deposit amount. This causes an arithmetic underflow when subtracting from `deposited_sol`, as `checked_sub` fails when the minuend is smaller than the subtrahend.

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

Copy Code

```
319     // Decrement deposited_sol state for the withdrawn amount
320     let current_withdrawal_deposit = ctx.accounts.pool_oracle.shares_to_assets(shares)?;
321     let state = &mut ctx.accounts.margin_lrt_state;
322     state.deposited_sol = state
323         .deposited_sol
324         .checked_sub(current_withdrawal_deposit)
325         .ok_or(LRTPoolError::InvalidAmount)?;
```

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

Copy Code

```
287     // Decrement deposited_sol state
288     let state = &mut ctx.accounts.margin_lrt_state;
289     state.deposited_sol = state
290         .deposited_sol
291         .checked_sub(withdrawal_shares)
292         .ok_or(LRTPoolError::InvalidAmount)?;
```

The expected behavior is to decrement `deposited_sol` by the proportional original deposit amount rather than the current asset value. For example, if a user deposited 10 SOL and withdraws 50% of their shares, `deposited_sol` should decrease by 5 SOL regardless of the current share price.

Users who deposit into the LRT pool through their margin accounts may be permanently blocked from withdrawing their funds after the share price appreciates. For example:

- A user who deposits 10 SOL at an exchange rate of 1.0, receives 10 shares.
- If the share price increases to 1.5 (50% appreciation), those 10 shares are now worth 15 SOL.
- When the user attempts to withdraw all shares, the system tries to subtract 15 SOL from the 10 SOL `deposited_sol` balance, causing an underflow and reverting with `InvalidAmount`.

This effectively locks user funds in the protocol with no way to withdraw, as any withdrawal attempt will fail. The issue compounds over time as the share price continues to appreciate, making it increasingly impossible for long-term depositors to exit their positions.

Proof of Concept

PoC Code

Copy Code

```
// Step 2: User B deposits 10 SOL at price 1.0
// deposited_sol = 10 SOL, gets ~10 shares
let deposit_amount = 10 * LAMPORTS_PER_SOL;
user_b
    .lrt_deposit(&lrt_pool.pool, deposit_amount, pool_fee_receiver_pubkey)
    .await?;

println!("User B deposited {} SOL at price 1.0", deposit_amount / LAMPORTS_PER_SOL);

// Step 3 & 4: Perform instant_withdrawal with manipulated exchange rate
// This bundles in one transaction:
// 1. update_oracle (to make oracle valid)
// 2. set_test_oracle_exchange_rate to 1.5 (simulating yield)
// 3. margin_instant_withdrawal (which will see rate 1.5 and fail)
//
// This should fail because:
// - deposited_sol = 10 SOL (recorded at deposit time)
// - shares_to_assets(10 shares) = 15 SOL (at new price 1.5)
// - 10 - 15 = UNDERFLOW
let fee_account = lrt_pool
    .pool
    .stake_mint
    .associated_token_address(&pool_fee_receiver_pubkey);

// Get user's deposit token balance (shares)
// Use margin_deposit_mint (LRT_MARGIN_DEPOSIT_MINT), not lrt_margin_deposit_mint_account
let deposit_mint = MintInfo::with_legacy(lrt_pool.pool.margin_deposit_mint(&airspace));
let user_deposit_account = deposit_mint.associated_token_address(user_b.tx.address());
let user_shares = ctx
    .rpc()
    .get_token_balance(&user_deposit_account)
    .await?
    .unwrap_or(0);

println!("User B has {} shares", user_shares);

let new_exchange_rate = 150_000_000u64; // 1.5 SOL per share
println!("Attempting instant_withdraw with exchange rate: {}", new_exchange_rate as f64 / 100_000

// Get stake price oracle for update_oracle call
let stake_price_oracle = derive_pyth_price_feed_account(
    env.ssol_oracle.pyth_feed_id().unwrap(),
    None,
    glow_test_service::ID,
);

// Try to withdraw all shares - should fail with InvalidAmount (underflow)
let result = lrt_pool
    .poc_margin_instant_withdrawal_with_rate_change(
        &ctx,
        &user_b_wallet,
        airspace,
```

```
            *user_b.tx.address(),
            stake_price_oracle,
            fee_account,
            pool_fee_receiver_pubkey,
            user_shares,
            new_exchange_rate,
        )
        .await;

    // Assert that it fails with InvalidAmount error due to underflow
    assert_custom_program_error(LRTPoolError::InvalidAmount, result);
```

Evidence

```
[2026-01-22T02:19:43.560512000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 invoke [1]
[2026-01-22T02:19:43.561159000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: UpdateOracle
[2026-01-22T02:19:43.563469000Z DEBUG solana_runtime::message_processor::stable_log] Program log: LP Balances: Assets 10.0, shares 10.0, stake 0.0
[2026-01-22T02:19:43.564172000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Oracle update. Assets 10.0, shares 10.0
[2026-01-22T02:19:43.564401000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Exchange rate 100000000
[2026-01-22T02:19:43.564451000Z DEBUG solana_runtime::message_processor::stable_log] Program data: igkz2+TGC5M4NK6XZgoHz3ebxTJzBQlO2QrXLdHS5tdOggnqYezJDXz9cXbnM3RzzuSiFc/ivMou9aPBxX+cq
JQbI0ymSPgKYA6ofgt6IVJXvmvlkJNtHB0kd64YOgtQL3z9q3QrEd8A4fUFAAAAAADh9QUAAAAAAOQLVAIAAAAA5AtUAgAAAAAAAAAAAAAAAPolxaQAAAAA+iXFpAAAAAA==
[2026-01-22T02:19:43.564654000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 consumed 42950 of 600000 compute units
[2026-01-22T02:19:43.564680000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 success
[2026-01-22T02:19:43.564705000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 invoke [1]
[2026-01-22T02:19:43.564999000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: TestSetOracleExchangeRate
[2026-01-22T02:19:43.565438000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Setting oracle exchange rate from 100000000 to 150000000
[2026-01-22T02:19:43.565616000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 consumed 5922 of 557050 compute units
[2026-01-22T02:19:43.565637000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 success
[2026-01-22T02:19:43.565665000Z DEBUG solana_runtime::message_processor::stable_log] Program GLoWMgcn3VbyFKiC2FGMgfKxYSyTJS7uKFwKY2CSkq9X invoke [1]
[2026-01-22T02:19:43.567559000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: AdapterInvoke
[2026-01-22T02:19:43.567673000Z DEBUG solana_runtime::message_processor::stable_log] Program data: zDPY5r/AZEsTr+TvnKsExaX5TR68KSFvbcD1uf79tHpF1qklVxWF8g==
[2026-01-22T02:19:43.568944000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 invoke [2]
[2026-01-22T02:19:43.570649000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Instruction: MarginInitWithdraw
[2026-01-22T02:19:43.576110000Z DEBUG solana_runtime::message_processor::stable_log] Program log: AnchorError occurred. Error Code: InvalidAmount. Error Number: 142021. Error Message:
Invalid amount.
[2026-01-22T02:19:43.576194000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 consumed 81753 of 535216 compute units
[2026-01-22T02:19:43.576210000Z DEBUG solana_runtime::message_processor::stable_log] Program LRtc6q4AhSr3k9dSLXpTRoAP1hBrgbQSiFkuQpuHaq3 failed: custom program error: 0x22ac5
[2026-01-22T02:19:43.576226000Z DEBUG solana_runtime::message_processor::stable_log] Program GLoWMgcn3VbyFKiC2FGMgfKxYSyTJS7uKFwKY2CSkq9X consumed 97665 of 551128 compute units
[2026-01-22T02:19:43.576240000Z DEBUG solana_runtime::message_processor::stable_log] Program GLoWMgcn3VbyFKiC2FGMgfKxYSyTJS7uKFwKY2CSkq9X failed: custom program error: 0x22ac5
POC SUCCESS: init_withdraw failed with InvalidAmount as expected (underflow)
test poc_deposited_sol_underflow_init_withdraw ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 1.10s
```

BVSS

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

Recommendation

It is recommended to track the proportional original deposit amount when decrementing `deposited_sol` during withdrawals.

Instead of using the current asset value from `shares_to_assets()`, calculate the proportional reduction based on the ratio of shares being withdrawn to total shares owned.

For `margin_instant_withdraw`:

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

Copy Code

```
319     // Calculate proportional original deposit to decrement
320     let user_total_shares = ctx.accounts.user_deposit_token_account.amount;
321     let proportional_deposit = state
322         .deposited_sol
323         .checked_mul(shares)
324         .ok_or(LRTPoolError::Overflow)?
325         .checked_div(user_total_shares)
326         .ok_or(LRTPoolError::Overflow)?;
327
328     state.deposited_sol = state
329         .deposited_sol
330
```

```
331             .checked_sub(proportional_deposit)
                .ok_or(LRTPoolError::InvalidAmount)?;
```

Apply the same pattern to `margin_init_withdraw`. This ensures that withdrawing X% of shares always reduces `deposited_sol` by X%, preventing underflow regardless of share price appreciation.

Remediation Comment

SOLVED: The **Blueprint Finance team** fixed the issue by implementing the suggested changes

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/921d07bd9739b21fad1fd205cda2946d59f766f6

7.3 WRONG ACCOUNT USED IN RENT BALANCE CHECK CAUSES MIGRATION FAILURES

// INFORMATIONAL

Description

The Glow LRT program includes a migration instruction `migrate_lrt_margin_authority` that expands the `LrtMarginAuthority` account to add a new `margin_fee_receiver` field.

During migration, the handler must ensure the account has sufficient lamports to cover rent after reallocation to a larger size.

As shown in the code snippet below, the handler calculates the rent shortfall by comparing the required rent against `ctx.accounts.pool.get_lamports()`.

However, the account being reallocated is `margin_authority`, not `pool`. This means the rent balance check uses the wrong account's lamport balance.

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

Copy Code

```
52  pub fn migrate_lrt_margin_authority_handler(ctx: Context<MigrateLrtMarginAuthority>) -> Result
53      // Check that the authority's size is what we expect (the old one)
54      let data_len = ctx.accounts.margin_authority.data_len();
55      assert_eq!(
56          data_len,
57          8 + std::mem::size_of::<LrtMarginAuthority>() - std::mem::size_of::<Pubkey>()
58      );
59      let lrt_authority_balance = ctx.accounts.pool.get_lamports();
60      let new_data_len = 8 + std::mem::size_of::<LrtMarginAuthority>();
61      let rent = Rent::get()?;
62      let required_rent = rent.minimum_balance(new_data_len);
63      if required_rent > lrt_authority_balance {
64          let shortfall = required_rent.saturating_sub(lrt_authority_balance);
65          ...
66      }
67      ...
68  }
```

The expected behavior is to check `ctx.accounts.margin_authority.get_lamports()` since that is the account being reallocated and needs sufficient rent-exempt balance.

The migration script fails because it does not correctly detect the amount needed for rent exemption.

By checking the wrong account's balance, the shortfall calculation will be incorrect, causing the reallocation to fail or not transferring the necessary lamports when needed.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (1.5)

Recommendation

It is recommended to use the correct account when checking the lamport balance for rent calculation, this is, using the `margin_authority` account for the corresponding calculation.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/0b99014519b2085dfe237e9b339e515a26140563

7.4 MISSING VALIDATION BETWEEN PROPOSER ACCOUNT AND STORED PROPOSER FIELD

// INFORMATIONAL

Description

The `accept_margin_fee_receiver_change` instruction closes the proposal account and refunds the rent to the `proposer` account.

As shown in the code snippet below, the `proposer` account is used in `close = proposer` but there is no constraint validating that it matches the `proposal.proposer` field stored in the proposal account.

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

Copy Code

```
#[derive(Accounts)]
pub struct AcceptMarginFeeReceiverChange<'info> {
    /// This must be the proposed fee receiver account
    pub signer: Signer<'info>,

    /// Rent refund destination; must match the proposer stored on the proposal
    #[account(mut)]
    pub proposer: AccountInfo<'info>,

    #[account(mut)]
    pub lrt_margin_authority: Box<Account<'info, LrtMarginAuthority>>,

    #[account(
        mut,
        seeds = [
            LRT_MARGIN_ACCOUNT_PROPOSAL_SEED,
            lrt_margin_authority.key().as_ref(),
        ],
        bump = proposal.bump,
        close = proposer,
        constraint = proposal.lrt_margin_authority == lrt_margin_authority.key() @ LRTPoolErro
        // The proposed account must sign to accept the change
        constraint = proposal.proposed_account == signer.key() @ LRTPoolError::InvalidProposed
    )]
    pub proposal: Account<'info, MarginAccountChangeProposal>,
}
```

The expected behavior is to validate that the `proposer` account matches `proposal.proposer`, ensuring the rent refund goes to the original proposer who paid for the account creation.

The rent refund could be sent to an incorrect address if a different account is passed as the `proposer` parameter.

This would result in the original proposer not receiving the rent they are entitled to from the closed proposal account.

BVSS

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

Recommendation

It is recommended to add a constraint validating that the provided `proposer` account matches the stored `proposal.proposer`.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/17224ae9ed163cfd9a0b5c1c47e16b32f4a9803d

7.5 MISSING MARGINACCOUNT TYPE VALIDATION IN CREATE_MARGIN_LRT_STATE

// INFORMATIONAL

Description

The `create_margin_lrt_state` instruction accepts `margin_account` as a `Signer<'info>` without validating that it is actually a valid `MarginAccount`.

As shown in the code snippet below, the account only needs to sign the transaction, but its type is not verified.

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

Copy Code

```
22 #[derive(Accounts)]
23 pub struct CreateMarginLrtState<'info> {
24     #[account(mut)]
25     pub payer: Signer<'info>,
26
27     pub margin_account: Signer<'info>,
28     // ...
29 }
```

All other margin instructions in the LRT program properly validate the `margin_account` using `AccountLoader<'info, MarginAccount>`:

- `margin_deposit.rs`
- `margin_instant_withdraw.rs`
- `margin_init_withdraw.rs`
- `margin_execute_withdraw.rs`
- `margin_cancel_pending_withdrawal.rs`
- `margin_refresh_lrt_position.rs`

No direct risk was identified since the created `MarginLrtState` would be unusable - all deposit and withdrawal operations properly validate the `MarginAccount` type.

However, the behavior is inconsistent with the expected system design, where only valid `MarginAccount` accounts should be able to create their corresponding `MarginLrtState`.

BVSS

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

Recommendation

It is recommended to use `AccountLoader<'info, MarginAccount>` with the `signer` constraint, consistent with other margin instructions:

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

Copy Code

```
#[derive(Accounts)]
pub struct CreateMarginLrtState<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(signer)]
    pub margin_account: AccountLoader<'info, MarginAccount>,
    // ...
}
```

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/80cab931335434589f740836598b0e03e8070db6

8. AUTOMATED TESTING

Description

Halborn used automated security scanners to assist with the 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 reviewers 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.

Results

ID	Package	Short Description
RUSTSEC-2024-0344	curve25519-dalek	Timing variability in curve25519-dalek's
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on ed25519-dalek

---