Assure DeFi®

THE VERIFICATION GOLD STANDARD

Security Assessment

RiftsProtocol

Date: 02/09/2025

Audit Status: FAIL

Audit Edition: Solana

ASSURE DEFI®

THE VERIFICATION GOLD STANDARD

Risk Analysis

Vulnerability summary

Classification	Description
🔴 High	High-level vulnerabilities can result in the loss of assets or manipulation of data.
🟠 Medium	Medium-level vulnerabilities can be challenging to exploit, but they still have a considerable impact on smart contract execution, such as allowing public access to critical functions.
🟡 Low	Low-level vulnerabilities are primarily associated with outdated or unused code snippets that generally do not significantly impact execution, sometimes they can be ignored.
🟢 Informational	Informational vulnerabilities, code style violations, and informational statements do not affect smart contract execution and can typically be disregarded.

Executive Summary

According to the Assure assessment, the Customer's smart contract is **Insecure.**

Insecure	Poorly Secured	Secured	Well Secured

Scope

Target Code And Revision

The audit was performed on the RIFTS protocol smart contracts, written in Rust using the Anchor framework and compiled to Solana BPF programs.

The scope included the following modules:

- programs/rifts-protocol/src/lib.rs
- programs/fee-collector/src/lib.rs
- programs/lp-staking/src/lib.rs
- programs/governance/src/lib.rs

Target Code And Revision

Project	Assure
Language	Rust
Codebase	https://github.com/riftsprotocol/programs/ Fixes v1: 295f02877390ef19bc6df88fbe8f9fdad146d21e
Audit Methodology	Static, Manual

AUDIT OVERVIEW

🟥🟥🟥 **HIGH**

1. Arbitrary price injection via UpdateJupiterOracle (Authorization & Data Source Integrity)

Location: programs/rifts-protocol/src/lib.rs

Issue: A malicious actor can:

Initialize their own JupiterPriceUpdate account (owned by this program).

Call update_jupiter_oracle using their arbitrary account as price_update, signing as any key for oracle_authority.

Overwrite the protocol's effective price with attacker-chosen values, if the handler consumes the price_update without enforcing publisher identity or registry membership.

Recommendation: Replace the account with a trusted oracle program (like, Switchboard/Pyth) and verify program ID and account state (slot freshness, confidence, etc.).

If you keep a program-owned buffer:

Add an on-chain registry: oracle_registry.authorized_publishers: Vec<Pubkey>.

In UpdateJupiterOracle, enforce:

```
require!(
  oracle_registry.authorized_publishers.contains(&oracle_authority.key()),
  ErrorCode::UnauthorizedOracle
);
// Also bind `price_update` with seed that includes oracle pubkey:
// seeds = [b"jup_price", rift.key().as_ref(), oracle_authority.key().as_ref()]
```

Never accept raw prices from unconstrained Account<...> without seed/program/owner checks.

2. Handcrafted Jupiter CPI with weak slippage & account-meta trust

Location: programs/fee-collector/src/lib.rs

Issue: The handler trusts arbitrary remaining_accounts to represent a valid Jupiter route. While Jupiter will run its own checks, you are accepting any route that the caller provides, without cross-checking its semantics.

minimum_amount_out is hardcoded to 95% of input, disregarding the minimum_out parameter supplied to the instruction. This dramatically weakens price protection and enables value leakage on stale routes or manipulated pools.

The binary format you assemble (try_to_vec of a local struct) does not match Jupiter route ABI, so in practice the call will fail if corrected later, the logic as-written would still be unsafe because it relies on caller-provided metas and a lax min-out.

Recommendation: Do not accept arbitrary Jupiter routes from callers:

Either precompute off-chain (governed signer) the exact Instruction (including full data and metas) and pass it as a verified CPI blob, or

Use Jupiter official CPI interface/SDK to construct routes and pin the exact expected accounts (and pool program IDs) in your handler.

Enforce exact slippage:

```
require!(minimum_out > 0, FeeCollectorError::InvalidMinimumOut);
require!(actual_amount_out >= minimum_out, FeeCollectorError::SlippageTooHigh);
```

Add a route allowlist (AMMs/pool programs) and reject any unrecognized program IDs in the metas.

3. Dangerous numeric truncation: multiple as u64 from wider integers

Location: programs/rifts-protocol/src/lib.rs

Issue: Casting from u128 to u64 can truncate silently in release builds, corrupting supply, price, or reward calculations.

Recommendation: Keep computations in u128 and downcast only with checked conversion:

```
use core::convert::TryFrom;
let safe = u64::try_from(value_u128).map_err(|_| ErrorCode::MathOverflow)?;
```

Consider fixed-point Q64.64 (or similar) for price/ratio math and also document scale factors.

4. Inconsistent PDA seeds for vault vs vault_authority

Location: programs/rifts-protocol/src/lib.rs and lp-staking/src/lib.rs

Issue: Seed inconsistency can cause authority mismatches and 'impossible to sign' situations, or worse accidental key collisions across different account roles if the same seeds are reused for different account types.

Recommendation: Standardize: for example

```
Token vault: ["vault", rift]
Vault authority: ["vault_auth", rift]
```

Add compile-time comments/tests asserting that PDAs differ and document the seed schema.

5. Token-account authority/owner checks are sparse

Location: Multiple #[derive(Accounts)] sections

Several Account<'info, TokenAccount> fields lack explicit constraints tying:

token_account.owner == expected_authority

token_account.mint == expected_mint

Some vaults are modeled as UncheckedAccount rather than Account<TokenAccount> (like collector_vault in fee-collector), relying on the SPL Token program to fail mismatched mints at runtime.

Issue: While SPL Token will block invalid transfers, business-logic assumptions (for example identity of the owner) are not formally enforced. This widens the attack surface (like approvals/delegates or stale associated accounts).

Recommendation: Prefer #[account(associated_token::...)] where possible.

Otherwise, enforce:

```
#[account(
  constraint = user_underlying.owner == user.key(),
  constraint = user_underlying.mint == underlying_mint.key()
)]
```

Replace UncheckedAccount vaults with Account<TokenAccount> plus mint/owner constraints.

6. Missing Mint Constraint in Voting

Location: programs/governance/src/lib.rs

Issue: In cast_vote, the voter token account is only checked for ownership, not that its mint matches governance.rifts_mint. An attacker can create a token with arbitrary supply and use it to inflate voting power.

Recommendation: Add a constraint in CastVote (and CreateProposal) to enforce:

```
constraint = voter_rifts_account.mint == governance.rifts_mint @
GovernanceError::InvalidRiftsMint
```

7. Ineffective Snapshot Protection

Location: programs/governance/src/lib.rs

Issue: Voting power is taken from the current token balance, not from a snapshot at proposal creation. The included VoteSnapshot struct is unused. This makes flash-loan style or post-proposal accumulation possible.

Recommendation: Implement true snapshot voting:

1. Create a VoteSnapshot PDA per voter at proposal creation.
2. Store voting power at snapshot.
3. In cast_vote, enforce votes use the stored snapshot power.

8. Under-Allocated Account Space

Location: programs/governance/src/lib.rs

Issue: Proposal and Governance accounts use std::mem::size_of::<T>() for allocation. These structs contain variable-length fields (String, Vec<u8>, Option), so runtime data can exceed allocated space, causing proposal creation or execution failures.

Recommendation: Define explicit INIT_SPACE constants with maximum field sizes,for example:

```
#[account(init, space = Proposal::INIT_SPACE)]
pub proposal: Account<'info, Proposal>;
```

Or split large/pending payloads into separate PDAs.

9. Hardcoded Decimals and Fixed Thresholds

Location: programs/governance/src/lib.rs

Issue: Thresholds (for example 1000 * 10^9 tokens to propose, 500k * 10^9 min participation) assume 9 decimals but never check the mint. If decimals differ, thresholds become meaningless. Also, fixed absolute numbers don't scale with supply changes.

Recommendation: Verify Mint.decimals == 9 at initialization.

Compute thresholds as percentages of total supply at snapshot (for example 20% participation).

10. Missing Vault Initialization

Location: programs/rifts-protocol/src/lib.rs

Issue: The vault PDA is derived and stored:

```
let (vault_pda, _) = Pubkey::find_program_address(vault_seeds, &crate::ID);
rift.vault = vault_pda;
```

but the PDA is never actually initialized as a TokenAccount. This breaks wrapping logic and creates a race condition where an attacker could pre-create the account and hijack the vault.

Recommendation: Initialize the vault TokenAccount during create_rift with:

```
#[account(init, token::mint = underlying_mint, token::authority = vault_authority)]
pub vault: Account<'info, TokenAccount>;
```

Enforce correct seeds and validate vault authority against PDA.

11. Unchecked External Program IDs in CPI Calls [Fixed ✅]

Location: programs/rifts-protocol/src/lib.rs

Issue: External programs are declared as UncheckedAccount<'info>:

pub fee_collector_program: UncheckedAccount<'info>,

pub lp_staking_program: UncheckedAccount<'info>,

Without program ID validation, an attacker can substitute a malicious program and take control of CPI execution, potentially draining funds or corrupting state.

Recommendation: Replace with strict program constraints:

```
#[account(address = FEE_COLLECTOR_PROGRAM_ID)]
pub fee_collector_program: Program<'info, FeeCollector>;
```

and similarly for lp_staking_program.

Manage upgrade authority of these programs via governance.

Fix: You now constrain fee_collector_program == fee_collector::ID and lp_staking_program == lp_staking::ID.

12. Token Account Authority Validation Missing

Location: programs/rifts-protocol/src/lib.rs

Issue: The vault authority PDA is derived with inconsistent seeds and validated only by PDA derivation, not by checking if it is actually the authority of the vault TokenAccount. This could allow mismatched vaults to slip through.

Recommendation: Standardize seeds (["vault_auth", rift.key().as_ref()]).

Enforce runtime validation:

```
require!(vault_token_account.owner == token_program.key());
require!(vault_token_account.authority == vault_authority.key());
```

13. Overflow in Fee Calculations

Location: programs/rifts-protocol/src/lib.rs

Issue: Fee calculation multiplies attacker-supplied fee_amount by burn_fee_bps:

```
let burn_amount = fee_amount
    .checked_mul(u64::from(rift.burn_fee_bps))?
    .checked_div(10000)?;
```

While checked math avoids silent overflow, maliciously large fee_amount can force a revert (DoS).

Recommendation: Bound inputs (fee_amount <= 1e18).

Use u128 for intermediate math:

```
let burn_amount = (u128::from(fee_amount) * u128::from(rift.burn_fee_bps) / 10_000)
    .try_into()
    .map_err(|_| ErrorCode::MathOverflow)?;
```

14. Hardcoded Jupiter Program ID

Location: programs/rifts-protocol/src/lib.rs

Issue: The Jupiter v6 router ID is hardcoded:

let jupiter_v6_id = "JUP6LkbZbjS1jKKwapdHNy74zcZ3tLUZoi5QNyVTaV4".parse::<Pubkey>()?;

If Jupiter rotates its deployment or the upgrade authority changes, swaps may fail or be unsafe.

Recommendation: Store whitelisted program IDs in a governance-controlled registry PDA.

Validate program_id against the registry at runtime.

15. Missing Mint Authority Validation

Location: programs/rifts-protocol/src/lib.rs

Issue: Calls like:

```
token::mint_to(mint_ctx, total_rewards)?;
```

assume the PDA is mint authority, but this relationship is never validated during initialization. If misconfigured, minting may fail or be hijacked.

Recommendation: On initialization, enforce:

```
require!(mint.mint_authority == COption::Some(pda.key()),
ErrorCode::InvalidMintAuthority);
```

Add integration tests verifying mint authority matches PDA.

16. Arithmetic Precision Loss in Oracle Calculations

Location: programs/rifts-protocol/src/lib.rs

Issue: Price averaging is done with plain integer division:

```
let avg_price = total_price.checked_div(count)?;
```

This truncates toward zero, introducing a systematic downward bias. Over time, this can be exploited for mispricing in mint/redeem operations.

Recommendation: Use fixed-point math (Q64.64) or scale division:

```
let avg_price = (total_price * 1_000_000u128 / count) as u64; // preserve 6 decimals
```

Track and enforce precision explicitly in oracle feeds.

17. Arithmetic Precision Loss in Oracle Calculations

Location: programs/lp-staking/src/lib.rs

Issue: The LP token vault and reward vault are declared as UncheckedAccount and are never initialized as SPL Token accounts.

They may exist as empty system accounts with no token data.

Transfers into them will fail at runtime.

If attacker pre-creates them, protocol vaults could be hijacked.

Recommendation: Replace with:

```
#[account(init, token::mint = lp_token_mint, token::authority = pool_authority)]
pub lp_token_vault: Account<'info, TokenAccount>;
#[account(init, token::mint = reward_token_mint, token::authority = pool_authority)]
pub reward_vault: Account<'info, TokenAccount>;
```

Ensure PDA authority is set and enforced during pool initialization.

18. Missing Mint Authority Validation (ClaimRewards)

Location: programs/lp-staking/src/lib.rs

Issue: The reward token mint is declared as:

```
pub reward_token_mint: Account<'info, Mint>,
```

but there is no validation that its mint authority equals the reward authority PDA. If misconfigured, minting will fail or mint could be controlled by another party.

Recommendation: Add constraint

```
#[account(
  constraint = reward_token_mint.mint_authority ==
COption::Some(reward_authority.key())
)]
pub reward_token_mint: Account<'info, Mint>;
```

Enforce this at initialization so rewards cannot be hijacked.

19. Integer Overflow in Reward Calculation

Location: programs/lp-staking/src/lib.rs

Issue: Reward calculation multiplies elapsed time and rewards per second:

```
let rewards = (safe_time_elapsed as u64)
    .checked_mul(pool.rewards_per_second)?
    .checked_mul(PRECISION)?; // PRECISION = 1e12
```

With PRECISION = 1e12, this multiplication always risks overflow in u64. Realistic parameters will cause runtime failure (DoS).

Recommendation: Use u128 for intermediate math:

```
let rewards = (safe_time_elapsed as u128)
```

```
    .checked_mul(pool.rewards_per_second as u128)?
    .checked_mul(PRECISION as u128)?;
```

Cap rewards_per_second and safe_time_elapsed at sane values.

20. Race Condition in Stake Function

Location: programs/lp-staking/src/lib.rs

Issue: The stake account uses init:

```
#[account(init, payer = user, ...)]
pub user_stake_account: Account<'info, UserStakeAccount>;
```

This will fail if the user has already staked once, preventing re-staking or compounding. Protocol becomes unusable for returning users.

Recommendation: Change to:

```
#[account(init_if_needed, payer = user, ...)]
pub user_stake_account: Account<'info, UserStakeAccount>;
```

Ensure logic resets or accumulates correctly on repeated stakes.

21. Precision Loss Leading to Reward Theft

Location: programs/lp-staking/src/lib.rs

Issue: Fixed-point math does division before multiplication:

```
let reward_per_share = rewards.checked_div(PRECISION)?.checked_mul(stake)?;
```

This truncates values, systematically underpaying users.

Recommendation: Always multiply before dividing to preserve precision:

```
let reward_per_share = (rewards as u128)
    .checked_mul(stake as u128)?
    .checked_div(PRECISION as u128)?;
```

Use u128 arithmetic for fixed-point math.

22. Missing Slippage Protection

Location: programs/lp-staking/src/lib.rs

Issue: If staking/unstaking interacts with external pools via CPI, there is no slippage or price impact protection. Attackers could sandwich users with MEV. If functions are purely token transfers, this issue is not exploitable.

Recommendation: If swaps occur: require minimum_out/max_slippage_bps parameters and enforce.

If only transfers: clarify in code comments that slippage is not applicable.

23. Unbounded Rewards Accumulation

Location: programs/lp-staking/src/lib.rs

Issue: accumulated_rewards_per_share grows monotonically in u64. Over time, it can overflow, bricking reward distribution.

Recommendation: Switch to u128 accumulator.

Add upper bounds or periodic compaction of accumulated values.

24. Missing Emergency Controls

Location: programs/lp-staking/src/lib.rs

Issue: No pause mechanism or emergency withdraw path exists. If exploitation occurs, governance cannot stop damage.

Recommendation: Add governance-controlled paused: bool flag checked in every entrypoint.

Provide an emergency withdraw function restricted to governance.

25. Reward Rate Update Vulnerability

Location: programs/lp-staking/src/lib.rs

Issue: Reward rate can be set to 0 or absurdly high values.

0 makes staking unprofitable (DoS).

Huge value drains treasury quickly.

Recommendation: Add bounds:

```
require!(new_rate > 0 && new_rate <= MAX_REWARD_RATE, ErrorCode::InvalidRate);
```

Define MAX_REWARD_RATE via governance.

26. Vote Delegation / Double Voting via Account Duplication

Location: programs/governance/src/lib.rs

Issue: A voter voting power is derived from the balance of a single token account passed into cast_vote.

The code does not verify that the same tokens are not reused across multiple accounts.

An attacker can split their RIFTS tokens across multiple accounts and sequentially transfer tokens to double-count votes.

Snapshot logic exists but is not enforced across all accounts, so this remains exploitable.

Recommendation: Enforce voting power from a single canonical token account per voter (for example an associated token account).

Alternatively, implement a true snapshot registry mapping voter pubkey to voting power at proposal creation, and reject votes from multiple token accounts.

27. Proposal Execution Without Effective Timelock

Location: programs/governance/src/lib.rs

Issue: The code checks:

```
current_time >= proposal.voting_end + governance.min_execution_delay
```

but min_execution_delay can be set to 0 during initialization. This allows immediate execution of proposals after voting ends, leaving no time for community or off-chain monitoring to react to malicious proposals.

Recommendation: Enforce a non-zero minimum at initialization (for example at least 24h).

Consider a governance parameter requiring supermajority or special authority to lower this delay.

28. Integer Overflow in Vote Counting

Location: programs/governance/src/lib.rs

Issue: Vote counting uses checked addition:

```
proposal.votes_for = proposal.votes_for
    .checked_add(voter_rifts_balance)
    .ok_or(GovernanceError::VoteOverflow)?;
```

While overflows are caught, if a voter’s balance is extremely large (close to u64::MAX), the addition will revert and cause a denial of service for that proposal. Multiple such votes can halt execution permanently.

Recommendation: Cap individual voting power at a safe bound (for example total supply).

Use u128 accumulators for vote counts and only downcast when emitting events.

29. Missing Signer Restriction in Proposal Execution

Location: programs/governance/src/lib.rs

Issue: The executor is declared as:

```
pub executor: Signer<'info>,  // Any signer can execute
```

Any signer can execute proposals once they pass, without being governance authority or proposer. While this doesn’t allow altering results, it enables griefing (front-running execution, spamming).

Recommendation: Restrict execution to governance authority or whitelisted executors.

Alternatively, allow open execution but add anti-spam logic (ex, once executed, mark proposal closed atomically).

30. Emergency Action Authority Bypass

Location: programs/governance/src/lib.rs

Issue: Emergency actions (pause protocol, freeze assets) are handled like normal proposals with no extra quorum or special approval. This means a minimal quorum proposal could permanently freeze or pause the protocol.

Recommendation: Require supermajority (>66%) for emergency proposals

Add an additional governance authority approval for Freeze/Pause actions.

Provide automatic expiry of emergency actions unless renewed.

31. PDA Seed Collision Risk [Fixed ✅]

Location: programs/governance/src/lib.rs

Issue: Proposal PDAs use seeds:

```
[b"proposal", governance.key().as_ref(), &governance.total_proposals.to_le_bytes()]
```

If total_proposals overflows u64::MAX, seeds wrap, causing PDA collisions. In theory, this could allow overwriting old proposals. While practically infeasible (2^64 proposals), it is a design flaw.

Recommendation: Use a larger counter (u128) for proposal IDs.

Or include a timestamp/slot in PDA seeds to prevent collisions.

Add checks against u64::MAX overflow.

Fix: You use checked_add so wrap causes error, not collision.

32. Governance parameter spoofing in execute_governance_proposal

Location: programs/rifts-protocol/src/lib.rs

Issue:The core entrypoint applies whatever new_* values the caller passes if the attached proposal is of type ParameterChange and already Executed. It does not bind those values to the payload approved on-chain (governance.pending_parameter_changes) nor verify proposal IDs/content.

Recommendation: In core, read and apply only governance.pending_parameter_changes when governance.parameter_change_proposal_id == proposal.id, then clear it. Reject if there's a mismatch or None.

33. Partner-fee redirection (unbound partner_vault)

Location: programs/rifts-protocol/src/lib.rs

Issue: process_fee_distribution sends partner_amount to any provided partner_vault when rift.partner_wallet.is_some(), but never proves that vault belongs to the configured partner or matches the mint.

Recommendation: Require partner_vault.owner == rift.partner_wallet.unwrap() and partner_vault.mint == vault.mint (or use ATA constraints).

34. LP supply overflow & bogus "sqrt" in wrap_tokens

Location: programs/rifts-protocol/src/lib.rs

Issue: lp_token_supply is set to (amount_after_fee as u128 * initial_rift_amount as u128) / 2 and then unchecked cast to u64.

Recommendation: Use a safe integer sqrt (u128) and checked downcast, or a sane formula with caps.

35. Accounting unit mix-ups

Location: programs/rifts-protocol/src/lib.rs

Issue: total_wrapped is increased by RIFT minted in wrap, but decreased by underlying redeemed in unwrap and get_pending_fees subtracts RIFTS-distribution counters from underlying fees.

Recommendation: Store and update units consistently (for example track both wrapped_underlying and minted_rift separately). Never cross-subtract different units.

36. Inconsistent PDA seeds for vault authority (DoS/brick funds)

Location: programs/rifts-protocol/src/lib.rs

Issue: Different instructions derive/sign vault authority with different seeds (["vault_auth", rift] in unwrap_tokens; ["vault", rift] in process_fee_distribution & jupiter_swap_for_buyback), and one place even sets authority seeds equal to the vault's seeds.

Recommendation: Pick one canonical seed set (ex, ["vault_auth", rift]) and use it everywhere (accounts constraints + signer seeds + docs). Never share seeds with the vault account itself.

37. Post-swap accounting is fictitious

Location: programs/fee-collector/src/lib.rs

Issue: process_fees sets rifts_bought = swap_amount (1:1) instead of reading the actual RIFTS tokens received.

Recommendation: Read token balances before/after, compute deltas, assert >= minimum_out.

38. CPI ABI/metas remain incorrect/fragile

Location: programs/fee-collector/src/lib.rs

Issue: Custom Jupiter layout (already noted before in this report) plus odd metas (adds router ID as a readonly meta ignores dex_program account). Even if ABI is fixed, this meta set is error-prone.

Recommendation: Use Jupiter's official CPI builder; pass exact expected metas; remove unused dex_program or enforce it.

39. Zero voting period allowed

Location: programs/governance/src/lib.rs

Issue: min_voting_period has no lower bound.

Recommendation: Enforce a minimum (like ≥ 24h) at initialization and/or proposal creation.

40. Vaults uninitialized at pool creation (functional blocker)

Location: programs/lp-staking/src/lib.rs

Issue: pool_lp_tokens and reward_vault are UncheckedAccount on init later used as Account<TokenAccount>.

Recommendation: Initialize these as SPL Token accounts at initialize_pool.

41. Reward mint authority not bound

Location: programs/lp-staking/src/lib.rs

Issue: Minting uses a PDA authority, but you never assert reward_token_mint.mint_authority == reward_authority.

Recommendation: Add constraints at init/claim.

◆◆◆ **MEDIUM**

1. Permissionless cleanup_stuck_accounts can grief (forced close)

Location: rifts-protocol/src/lib.rs, function at ~1225, accounts at ~1779

Issue: While fund theft is prevented by PDA matching, this enables griefing (third parties can close resources you might prefer to keep for debugging).

Recommendation: Require creator to sign, or gate via governance. Alternatively, add a cool-down window or event-based allowlist for cleanups.

2. Rate limiting comment without enforcement

Location: fee-collector/src/lib.rs (fee collection)

Issue: Code obtains Clock::get()?.unix_timestamp with a comment "max 10 per hour" but does not persist or enforce a counter.

Recommendation: Track (window_start, window_count) in state and require!(window_count < 10) within the current hour.

3. Ineffective / Brittle Logic Checks

Location: programs/governance/src/lib.rs

Issue: TokenBalanceDecreased check in voting is ineffective (always passes on first vote).

Fixed participation threshold ignores future supply changes.

governance.authority is a single key (no multisig).

Recommendation: Remove or redesign ineffective checks.

Use percentage-based participation.

Replace single authority with multisig PDA for robustness.

4. Invalid Jupiter ABI in Fee-Collector Jupiter CPI

Location: fee-collector/src/lib.rs

Issue: The program defines a custom struct:

```rust
#[derive(AnchorSerialize, AnchorDeserialize)]
struct JupiterSwapData {
    amount_in: u64,
    minimum_amount_out: u64,
    platform_fee_bps: u16,
}
```

and serializes it via .try_to_vec() before calling Jupiter:

```rust
let data = JupiterSwapData { ... }.try_to_vec()?;
let ix = Instruction {
    program_id: JUPITER_ROUTER_ID,
    accounts: jupiter_swap_accounts.to_vec(),
    data,
};
```

This does not match Jupiter v6 router's expected instruction layout, which includes additional fields and a different serialization format. As a result, the CPI will always fail at runtime with "invalid instruction data," preventing buyback execution.

While this bug prevents the slippage and arbitrary route acceptance issues from being exploitable today, once the ABI is corrected those higher-severity vulnerabilities become active.

Recommendation:

Replace the custom struct with Jupiter's official instruction schema (via their SDK or verified CPI interface).

Use the exact instruction layout and serialization Jupiter expects (including all required fields).

Add integration tests on localnet to confirm that CPI calls succeed end-to-end with realistic routes.

Once corrected, immediately also apply the slippage and account-validation fixes, otherwise the function will expose critical vulnerabilities.

5. Timestamp Manipulation Vulnerability

Location: programs/lp-staking/src/lib.rs

Issue: Rewards depend on Clock::get()?.unix_timestamp, which validators can manipulate slightly (ex, by a few seconds). This allows small reward distortions.

Recommendation: Tolerate small deviations by bounding safe_time_elapsed

For higher trust, consider oracle-based block time or use slot-based accounting.

6. Reentrancy Risk in Proposal Execution [Fixed ✅]

Location: programs/governance/src/lib.rs

Issue: Proposal status is set to Executed before all effects complete. If future versions integrate external CPIs (ex, treasury transfers, upgrades), this could enable reentrancy attacks where state is inconsistent during external calls.

Recommendation: Apply the "checks-effects-interactions" pattern: update state only after external calls succeed.

Add a simple reentrancy guard flag to prevent recursive execution.

Fix: Current execute path only writes governance state; no external CPI side-effects here.

7. Fragile Jupiter CPI: signer not guaranteed in metas

Location: programs/rifts-protocol/src/lib.rs

Issue: jupiter_swap_for_buyback signs with vault_authority seeds, but the instruction metas are only from remaining_accounts. If the route metas don't include vault_authority (as a signer), your signature won't apply and the CPI fails.

Recommendation: Assert that remaining_accounts contains the expected PDA (as signer), or construct/verify the metas yourself.

8. close_rift compares to the wrong "system program" key

Location: programs/rifts-protocol/src/lib.rs

Issue: Compares rift.vault to Pubkey::default() while the System Program is 11111111111111111111111111111111.

Recommendation: Compare with system_program::ID.

9.calculate_price_deviation casts u64 to u16 unchecked

Location: programs/rifts-protocol/src/lib.rs

Issue: deviation as u16 can truncate on large deviations.

Recommendation: Bound-check before cast (for example, require!(deviation <= u16::MAX as u64, …)), or store as u32.

10.Reentrancy guard applied late in unwrap_tokens

Location: programs/rifts-protocol/src/lib.rs

Issue: Guard is set after burning/transfers.

Recommendation: Set guard immediately after entry; clear only at the end.

11.“Fee distribution” does not actually burn/swap

Location: programs/rifts-protocol/src/lib.rs

Issue: process_fee_immediately updates counters but doesn’t execute burns or swaps.

Recommendation: Either execute the actions or rename/state these as “planned” & reconcile on actual execution.

12. collector_vault never initialized (functional blocker)

Location: programs/fee-collector/src/lib.rs

Issue: Declared UncheckedAccount in init, later used as Account<TokenAccount>.

Recommendation: Initialize with #[account(init, token::mint = <underlying>, token::authority = collector_authority, …)]

13. Slippage check not enforced on-chain result

Location: programs/fee-collector/src/lib.rs

Issue: You compute a slippage bound but don’t verify post-swap balances against minimum_out.

Recommendation: After swap, assert received >= minimum_out and revert otherwise

14. Precision/overflow hazards (u64 + PRECISION=1e12)

Location: programs/lp-staking/src/lib.rs

Issue: rewards * PRECISION in update_pool_rewards can overflow u64 under realistic params.

Recommendation: Do math in u128 (time * rate * PRECISION), cap inputs, then checked downcast.

◆ ◆ ◆ LOW

1. Hardcoded program IDs without environment guards

Location: build_jupiter_swap_instruction

Issue: Jupiter router ID hardcoded. Fine for mainnet, but hazardous across clusters.

Recommendation: Gate by cluster or store in governance-controlled config & assert against a whitelist.

2. Unused/irrelevant accounts

Location: programs/fee-collector/src/lib.rs

Issue: dex_program is provided, never used.

Recommendation: Remove or enforce/address match.

◇ ◆ ◆ INFORMATIONAL

1. DeFi CPI Safety

Recommendation: Pin pool program IDs and token program ID at every CPI site.

For routing/aggregation (Jupiter), ingest fully-formed instructions from a trusted signer path and do not reconstruct from caller metas.

Enforce strict min-out at the handler boundary.

2. Oracle Integrity

Recommendation: Prefer Pyth/Switchboard verified feeds (check exponent/price/confidence, max staleness, and slot).

If custom buffers remain, bind with seeds that include the publisher key, and tie publisher to governance registry.

3. Numerics

Recommendation: Keep u128 internally & use checked downcasts only where inevitable.

Adopt fixed-point helpers with unit tests for rounding.

4. Accounts & Constraints

Recommendation: Encode ATAs via associated_token constraints / otherwise explicit owner/mint checks.

Normalize PDA seeds per account role and add tests asserting derived keys.

5. Pausing

Recommendation: Use a guard flag around all state mutations and strictly enforce pause on mint/burn/transfer and price writes.

Technical Findings Summary

Findings

Vulnerability Level		Total	Pending	Not Apply	Acknowledged	Partially Fixed	Fixed
🟥	HIGH	41					2
🟧	MEDIUM	14					1
🟨	LOW	2					
🟩	INFORMATIONAL	5					

Assessment Results

Score Results

Review	Score
Global Score	**30/100**
Assure KYC	Not completed
Audit Score	30/100

The Following Score System Has been Added to this page to help understand the value of the audit, the maximum score is 100, however to attain that value the project must pass and provide all the data needed for the assessment. Our Passing Score has been changed to 84 Points for a higher standard, if a project does not attain 85% is an automatic failure. Read our notes and final assessment below. The Global Score is a combination of the evaluations obtained between having or not having KYC and the type of contract audited together with its manual audit.

<u>Audit FAIL</u>

The solana programs audit has identified critical vulnerabilities. As a result, the audit has not passed. All identified issues must be resolved and re-audited before the contract can be considered secure for production use.

Disclaimer

Assure Defi has conducted an independent security assessment to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the reviewed code for the scope of this assessment. This report does not constitute agreement, acceptance, or advocating for the Project, and users relying on this report should not consider this as having any merit for financial advice in any shape, form, or nature. The contracts audited do not account for any economic developments that the Project in question may pursue, and the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude, and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are entirely free of exploits, bugs, vulnerabilities or deprecation of technologies.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence, regardless of the findings presented. Information is provided ‘as is, and Assure Defi is under no covenant to audit completeness, accuracy, or solidity of the contracts. In no event will Assure Defi or its partners, employees, agents, or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions or actions with regards to the information provided in this audit report.

The assessment services provided by Assure Defi are subject to dependencies and are under continuing development. You agree that your access or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies with high levels of technical risk and uncertainty. The assessment reports could include false positives, negatives, and unpredictable results. The services may access, and depend upon, multiple layers of third parties.