ASSURE DEFI®

THE VERIFICATION GOLD STANDARD

SECURITY ASSESSMENT REPORT

NAME:	VOLTAIC
STATUS:	PASS
DATE:	27/12/2025

Risk Analysis

Vulnerability summary

Classification	Description
🔴 High	High-severity issues can lead to **direct loss of funds**, **unauthorized state changes**, or **permanent corruption of on-chain data**. These vulnerabilities may allow attackers to drain program-owned accounts, bypass signer or ownership checks, or arbitrarily manipulate critical program logic.
🟠 Medium	Medium-severity issues are generally **more difficult to exploit** or require specific conditions, but they can still **negatively affect program security or correctness**. Examples include insufficient account validation, missing constraints, or logic flaws that could enable unintended behavior under certain circumstances.
🟡 Low	Low-severity issues typically relate to **best-practice deviations**, **inefficient logic**, or **edge-case behavior** that does not immediately threaten funds or program integrity. These findings generally have minimal impact on execution but may reduce code robustness or maintainability.
🟢 Informational	Informational findings include **code style issues**, **unused variables or instructions**, **documentation gaps**, or **general recommendations**. These do not affect program security or execution and are provided solely to improve code clarity and long-term maintainability.

Executive Summary

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

Insecure	**Poorly Secured**	**Secured**	**Well Secured**

Scope

Target Code And Revision

For this audit, we performed research, investigation, and review of the Voltaic verifying both the functional logic and surface-level implementation of the program, with access to the underlying source code.

Target Code And Revision

Project	Assure
Language	Rust
Codebase	https://github.com/Voltaic-Sol/Voltaic/blob/main/lib.rs Commit: 59b53970bc14c151fedc3b65549eab767bb4f9ca Fixed version [commit]: 58b452e330056cbfdf8af27d8728c55665e033e1 Deployed address: https://solscan.io/account/FCrSSe5yTaL8svSdnBxFoDexWBv1gbGcJrNCF7gtE3UT
Audit Methodology	Static, Manual

AUDIT OVERVIEW

◆◆◆ HIGH

1. Whitelist entries can be removed by anyone (DoS + rent theft) [Fixed ✅]

Issue:

Whitelist_remove handler does no authorization and returns Ok(()) immediately.

```
Line 117: pub fn whitelist_remove(_ctx: Context<WhitelistRemove>) -> Result<()> { }
```

WhitelistRemove account context closes the whitelist PDA to authority, which is any signer.

```
Lines 687-702: close = authority, seeds = [WL_SEED, pool, wallet].
```

This is a straight access-control failure on a permissioned gating mechanism.

Recommendation:

Add admin enforcement in either the handler or account constraints (ideally both):

Handler fix (minimal):

- Call only_admin(&ctx.accounts.pool, &ctx.accounts.authority)?; at the start of whitelist_remove.

Accounts fix (better UX, earlier fail):

Add a constraint to WhitelistRemove:

- constraint = pool.admin == authority.key() @ DexError::Unauthorized

Also consider:

- requiring the removed whitelist PDA to match (pool, wallet) and be owned by your program (defense-in-depth).

Fix: The handler enforces admin: only_admin(&ctx.accounts.pool, &ctx.accounts.authority)?; (around lines 141–145).

The accounts struct also enforces it: constraint = pool.admin == authority.key() inside WhitelistRemove (around 695–700). Non-admins can't remove other whitelist entries or steal the rent.

2. lp_vault unconstrained in AddLiquidity lets first LP steal MIN_LP_LOCK and “unlock” the pool [Fixed ✅]

Issue:

in initialize_pool, lp_vault is correctly created as the ATA owned by pool_authority.

```
Lines ~642-650: associated_token::authority = pool_authority
```

In AddLiquidity, lp_vault is only:

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

```
Line ~752 there is no ATA constraint, no "must be owned by pool_authority", no mint
check, no linkage in pool state.
```

In add_liquidity, on first mint (total_lp_before == 0) you mint MIN_LP_LOCK to lp_vault:

Function add_liquidity around the first-mint branch, mint_to_any_signed(..., &ctx.accounts.lp_vault, ..., MIN_LP_LOCK, ..)

This **Breaks core AMM invariant assumption**: pool cannot be fully drained/reset (the reason MIN_LP_LOCK exists).

Enables full liquidity removal = makes the pool vulnerable to:

- reserve imbalances / “donation theft” edge cases
- supply-reset behaviors that many AMMs intentionally prevent

Undermines economic safety and any downstream integrations assuming Uniswap-V2-style minimum liquidity lock.

Recommendation:

In AddLiquidity, constrain lp_vault to the canonical ATA:

```
#[account(
  mut,
  associated_token::mint = lp_mint,
  associated_token::authority = pool_authority,
  associated_token::token_program = lp_token_program
)]
pub lp_vault: Box<InterfaceAccount<'info, TokenAccount>>;
```

Even stronger:

- Store lp_vault pubkey in Pool state and enforce has_one = lp_vault.
- Additionally enforce pool_authority is the correct PDA (see mediumfinding).

Fix: lp_vault is constrained to be the canonical ATA for (lp_mint, pool_authority) via associated_token::mint = lp_mint + associated_token::authority = pool_authority (around 764–772).

pool enforces has_one = lp_vault (around 734–742) and Pool now stores lp_vault: Pubkey (line 961).

Now the program will only mint the locked LP to the pool’s vault, not an attacker-controlled account.

3. Token-2022 extensions can break user guarantees (slippage) and/or silently tax users/LPs [Partially Fixed ✅]

Issue:

Program explicitly supports Tokenkeg and Token-2022:

```
is_supported_token_program() checks spl_token::ID and spl_token_2022::ID. Lines
1004-1006
```

Swap slippage protection checks internal computed amount_out, not necessarily actual tokens received by the user:

```
require!(amount_out >= min_out, DexError::SlippageExceeded); in swap_exact_in.
```

User protection bypass (min_out not guaranteeing actual received amount).

LP value leakage if fees are taken from pool outputs.

Integration fragility: pools may behave “correctly” per math but “incorrectly” per user expectations.

Recommendation:

Safest: disallow Token-2022 mints with TransferFee / similar extensions at initialize_pool.

Better UX: if allowing transfer-fee tokens:

- compute expected fee from mint config and enforce net_received >= min_out, or
- measure user balance before/after the transfer-out and enforce delta >= min_out (atomic revert is safe on Solana).

Fix: The program now use actual vault deltas for inputs:

swap_exact_in: transfers in, then reload() vaults and computes actual_in = vault_after - vault_before (so if the input token takes a fee, the AMM math uses what the pool actually received). (lines 470–550)

add_liquidity: same approach; uses actual_a_in, actual_b_in based on vault deltas. (lines 210–320)

Slippage checks still compare computed amount_out to min_out before transferring out:

swap_exact_in: require!(amount_out >= min_out, ...) happens before the transfer-out. (lines 560–570)

If the output token has TransferFeeConfig, the user can receive less than min_out, yet the check passes.

remove_liquidity: min_out_a/min_out_b are checked against computed amounts, not the net received post-transfer. (lines 360–420).

To fully fix it, you still need either:

pre/post user balance delta checks for the output token account(s), or

compute expected transfer fee and enforce net_received >= min_out.

4. Token-2022 required extensions can brick pools (DoS) [Fixed ✅]

Issue:

All CPIs use token_interface::transfer_checked with no mechanism to pass required "remaining accounts" for Token-2022 extensions (for exampl, TransferHook's extra accounts list).

Recommendation:

During initialize_pool, explicitly inspect mint extensions and reject unsupported configurations, or extend CPIs to support required remaining accounts (harder, and requires a careful allowlist).

Fix: Added validate_token_2022_mint() and call it during initialize_pool for mint A/B/LP mint when the token program is Token-2022 (lines 64–92)

In validate_token_2022_mint() program explicitly reject extensions that typically require extra accounts / special handling:

TransferHook, ConfidentialTransfer*, DefaultAccountState, InterestBearingConfig (lines 1165–1174)

and also deny-by-default any unknown extension (future-proof).

So "pool bricked because transfers need remaining accounts" is largely addressed by not allowing those mints.

5. emergency_drain can rug all pool reserves [Fixed ✅]

Issue:

emergency_drain allows admin to transfer arbitrary amount from vault A or B to any destination accounts.

Guard is only only_admin

Recommendation:

If you want credible non-custodial posture:

- remove emergency_drain, or
- gate it behind a timelock, multisig, and require paused_swaps && paused_liquidity for N slots before enabling.

Fix: There is no emergency_drain instruction (no "emergency"/"drain" symbol at all), so the rug-lever is gone in this version.

◇ ◇ ◆ **MEDIUM**

1. Missing PDA seed constraints for pool_authority (defense-in-depth) [Fixed ✅]

Issue:

Most contexts accept:

```
/// CHECK: PDA authority signer
pub pool_authority: UncheckedAccount<'info>,
```

and only constrain:

pool.authority == pool_authority.key().

It is important to note that today it’s probably safe because pool.authority is set at init and not mutable by users. But it’s weaker than necessary and increases blast radius if any future refactor introduces a pool-authority mutability bug.

Recommendation:

In all instructions, enforce:

```
#[account(
  seeds = [AUTH_SEED, pool.key().as_ref()],
  bump = pool.bump_authority
)]
pub pool_authority: UncheckedAccount<'info>;
```

Fix: pool_authority is now enforced with PDA seeds + bump in:

InitializePool (line 607)

AddLiquidity (lines 743–751)

RemoveLiquidity (lines 825–834)

SwapExactIn (lines 893–902)

So it’s no longer just “unchecked + equals stored pubkey” now it’s actually constrained to the PDA derivation.

LOW

1. MEV / sandwich exposure [Acknowledge]

Issue:

This AMM is intrinsically exposed to:

- sandwiching of swap_exact_in
- JIT liquidity / backrun strategies
- toxic flow during volatile periods

This is not a correctness bug, but it is a real economic attack surface.

Recommendation:

Encourage users to use tight min_out (already supported)

Consider integrating with private orderflow / bundles (Solana ecosystem)

Add optional “price limit” style constraints (like max price impact)

INFORMATIONAL

No informational issues were found.

Technical Findings Summary

Findings

Vulnerability Level		Total	Pending	Not Apply	Acknowledged	Partially Fixed	Fixed
🟥	HIGH	5				1	4
🟧	MEDIUM	1					1
🟨	LOW	1			1		
🟩	INFORMATIONAL	0					

Assessment Results

Score Results

Review	Score
Global Score	**85/100**
Assure KYC	Not Completed
Audit Score	85/100

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.

Audit PASS

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.

After the development team's review, all critical vulnerabilities have been addressed/reviewed, and the audit results are satisfactory.

Disclaimer