InshAllah - Stablecoin Security review

Version 1.0

Table of Contents

1 About Egis Security

Egis Security is a team of experienced smart contract researchers, who strive to provide the best smart contract security services possible to DeFi protocols.

The team has a proven track record on public auditing platforms like Code4rena, Sherlock, and Cantina, earning top placements and rewards exceeding $250,000. They have identified over 300 high and medium-severity vulnerabilities in both public contests and private audits.

2 About InshAllah - Stablecoin

iAUSD is an algorithmic stablecoin implemented on Solana that utilizes a Collateralized Debt Position (CDP) model backed by iASOL as collateral. This implementation draws from extensive research of successful stablecoin architectures including DAI (MakerDAO), RAI (Reflexer), and FRAX, adapting their proven mechanisms to the Solana ecosystem while incorporating protocol-specific optimizations.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	InshAllah - Stablecoin
Repository	Private
Commit hash	0c37a3faf1046ecec1bed6046863496402c8c5d9
Resolution	7d3add598688c06fb4640aa7f2efb7a52bf3ef6f
Documentation	Private
Methods	Manual review

Scope

contract/programs/contract/src/instructions/*
contract/programs/contract/src/constants.rs
contract/programs/contract/src/errors.rs
contract/programs/contract/src/events.rs
contract/programs/contract/src/lib.rs
contract/programs/contract/src/state.rs
contract/programs/contract/src/utils.rs

Issues Found

Critical risk	0
High risk	1
Medium risk	4
Low risk	3
Informational	0

6 Findings

6.1 High risk

6.1.1 Pyth Oracle util logic doesn't normalize the feed price

Severity: *Medium risk*

Context: utils.rs#L10-L47

Description: The SOL/USD price feed from Pyth (H6ARHf6YXhGYeQfUzQNGk6rDNnLBQKrenN712K4AQJEG) uses an exponent of -8, meaning a price of $153.563 is represented as 15356300000 (with 8 decimal places).

In the current `get_sol_price` implementation (used in mainnet deployment), this value is returned without scaling, exposing the protocol to incorrect price usage in downstream logic. This affects critical operations like health factor calculation during minting and collateral withdrawal, potentially allowing mispriced transactions.

This results in compromised health calculation, which may result in minting unbacked iAUSD.

Recommendation: Update `get_sol_price` to accept a target precision (as a uint) and scale the oracle result accordingly. This ensures consistent and accurate value interpretation across all use cases.

Resolution: Fixed.

6.1.2 Health calculation assume 1:1 sol:iASOL

Severity: *Medium risk*

Context: state.rs#L102-L108

Description: `iASOL` is a Liquid Staking Token (LST) that accrues yield on staked SOL over time. As a result, the value of `iASOL` increases relative to `SOL`, meaning 1 `iASOL` will eventually be worth more than 1 `SOL`. At the time of writing, the conversion rate is approximately `1 SOL = 0.987718 iASOL`.

However, the protocol currently treats `iASOL` at a 1:1 value with SOL in collateral accounting. This leads to underestimating the value of user collateral, since the protocol assumes `iASOL` is worth less than it actually is. Also, when admin calls `extract_yield` this will always result in position health decrease, if we assume the SOL price has not changed.

Recommendation: Introduce a scaling ratio parameter in the protocol's config, representing the current exchange rate between SOL and iASOL. This ratio should be: - Updated periodically (e.g., weekly) via an on-chain admin or oracle mechanism.

This approach ensures both protocol safety and fairness for users holding appreciating LSTs like iASOL.

Resolution: Fixed.

6.2 Medium risk

6.2.1 `extract_yield` may put user position into liquidatable state

Severity: *Medium risk*

Context: extract_yield.rs#L70-L106

Description: `extract_yield` function may be called by admin for any user position regardless it's health factor once every 2 days. This will result in withdrawing 10% of the user collateral, which may result in decreasing it's position health to a liquidatable state:

```
require!(
      current_epoch > position.last_yield_extraction_epoch,
      ErrorCode::YieldAlreadyExtractedThisEpoch
  );
  // For MVP, use a fixed 10% yield extraction rate
  let yield_amount = position.collateral_amount
      .checked_mul(10)  // 10% extraction rate
      .ok_or(ErrorCode::MathOverflow)?
      .checked_div(100)
      .ok_or(ErrorCode::MathOverflow)?;

  // Only extract if there's a meaningful amount
  require!(yield_amount > 0, ErrorCode::YieldAmountTooSmall);
  // Only proceed if position has collateral
  require!(
      position.collateral_amount >= yield_amount,
      ErrorCode::InsufficientCollateral
  );
  // Transfer yield to collection wallet
  let seeds = &[
      POSITION_SEED,
      position.owner.as_ref(),
      &[position.bump],
  ];
  let position_signer = &[&seeds[..]];
  let cpi_accounts = Transfer {
      from: ctx.accounts.collateral_vault.to_account_info(),
      to: ctx.accounts.yield_collection_account.to_account_info(),
      authority: position.to_account_info(),
  };
  let cpi_context = CpiContext::new_with_signer(
      ctx.accounts.token_program.to_account_info(),
      cpi_accounts,
      position_signer,
  );
  token::transfer(cpi_context, yield_amount)?;
```

This combined with the fact that the system assumes `1:1 sol:iASOL` rate always result in decreasing users position health.

Recommendation: Panic, if the user position enters `Warrning` state and decrease the available amount to withdrawn for each epoch.

Resolution: Fixed by changing the amount to be withdrawn + fixing M1.

6.2.2 Missing epochs results in missed yield

Severity: *Medium risk*

Context: extract_yield.rs#L58-L68

Description: Currently `extract_yield` retrieves a fixed % of yield from a position. The function can only be called for previous epochs, meaning it has to be called every single epoch for every single position.

The issue is, if an epoch is missed for a position, then that yield is lost, as `last_yield_extraction_epoch` is always updated to `current_epoch` and is collected once, it doesn't matter if there are more than 1 epochs that are available to be claimed.

This might not be an issue in the beginning of the protocol, as it will naturally have less positions, but as it grows there will be more users and more positions, so it's vital to correctly `extract_yield` for all positions.

Not calling `extract_yield` for an epoch, results in loss of yield for that epoch

Recommendation: Collect yield for all the epochs have passed correctly, not just for 1.

Resolution: Fixed by extracting the yield for all missed epochs based on the latest collateral value.

6.2.3 Admin private key is being hardcoded in the test file

Severity: *Medium risk*

Context: extract_yield.rs#L58-L68

Description:

In the file contract/tests/contract.ts, a hardcoded private key is present for an `sabpHDMCJfXtfbr6TNbEmC5WMoHNWE2DAr` used in tests. While it may be intended only for local development or CI environments, this practice poses significant security risks.

We assume that this is the account used for Mainnnet also because `init` instruction enforce the admin account matches the given pubkey.

If the repository is ever made public—intentionally or by oversight—this private key could be exposed, enabling attackers to assume control of the associated address. Even if the address is not currently used in production, this oversight could lead to key reuse vulnerabilities, especially if the same key is inadvertently used elsewhere in the codebase. If someone malicious takes control, he can execute admin controlled functions, or even update the contract bwtecode.

Recommendation: Remove all hardcoded private keys from the codebase. Use environment variables or secure key management tools (e.g., HashiCorp Vault, AWS Secrets Manager) to inject test keys at runtime.

Resolution: Fixed by removing the key and using different for Mainnet deployment.

6.3 Low risk

6.3.1 `uri` in metadata points to solana example

Severity: *Low risk*

Context: initialize.rs#L110-L111

Description: When initializing gUSD (iAUSD) metadata in `init` inx, the constructed `DataV2`, which is passed to `create_metadata_accounts_v3` holds a hardcoded example uri:

```
    let data = DataV2 {
        name: "gUSD Stablecoin".to_string(),
        symbol: "gUSD".to_string(),
        uri: "https://raw.githubusercontent.com/solana-developers/program-examples/
            new-examples/tokens/tokens/.assets/spl-token.json".to_string(),
        seller_fee_basis_points: 0,
        ...
```

Recommendation: Use the correct `uri` instead of the solana example.

Resolution: Partially fixed by filling the uri with empty spaces. Team will edit it once they have their logo designed.

6.3.2 User can open a lot of dust positions to force the admin to pay for calling `extract_yield`

Severity: *Low risk*

Context: founder_dao/src/instructions/start_presale.rs

Description: In order to collect yield, the protocol uses `extract_yield` which collects yield for the specified position. This can become a more intensive and long operation, the more positions there are as `extract_yield` has to be called for each.

Since users can create a position without any restrictions, a user can create many dust positions in order to make the extraction of yield even more cumbersome.

As a result the extraction of yield becomes more cumbersome and more gas intensive for the protocol.

Recommendation: We recommend an initial minimum deposit to the position, in order to make creating a massive amount of positions more capital intensive, thus deterring users from spamming dust positions.

Resolution: Fixed.

6.3.3 No grace period for when system is reactivated, can lead to unfair liquidations

Severity: *Low risk*

Context: reactivate_system.rs#L22-L32

Description: The system can be deactivated and then activated again. In that period of time users cannot deposit any collateral to improve the health of their positions, meaning that during inactive

period the health of users positions can drop, they cannot be liquidated during the inactive period, but the moment the system is reactivated they are instantly eligible to be liquidated, which is unfair to all users that wanted to keep their positions open, but couldn't increase the health of their positions.

This may result in unfair liquidations to users that couldn't deposit collateral.

Recommendation: Track when the system was activated again and add a 30 minute grace period in which liquidations cannot happen.

Resolution: Acknowledged.