yAudit Asymmetry dASF Review

Review Resources:

None beyond the code repositories.

Auditors:

- HHK
- adriro

Table of Contents

Review Summary

Asymmetry dASF

The dASF protocol enables users to purchase ASF tokens using USAD at a discounted rate. Forked from Yearn's dYFI, dASF holders can redeem ASF at a discount determined by the duration they choose to lock ASF into veASF. Purchased tokens are paid for using Asymmetry's stablecoin, USAD. The codebase also includes a utility contract that allows users to exchange dASF for ASF using a flashloan to obtain the required USAD.

The contracts of the dASF repository were reviewed over three days. Two auditors performed the code review between April 7th and April 9th, 2025. The repository was under active development during the review, but the review was limited to the latest commit e41a1c6b5b8bb7dc8d54de134374f50fd49b4582.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
src
├── dASF.sol
├── flash_dump.vy
├── interfaces
│   ├── AggregatorV3Interface.vyi
│   ├── IBalancerVault.vyi
│   ├── ICurveCryptoPool.vyi
│   ├── ICurveStablePool.vyi
│   ├── IDASF.vyi
│   ├── IFlashLoanRecipient.vyi
│   ├── IRedemption.vyi
│   └── IVEASF.vyi
├── ownable_2step.vy
├── price_feed.vy
└── redemption.vy
```

After the findings were presented to the Asymmetry team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Asymmetry and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	The Ownable2Step logic in Solidity and Vyper variants provides suitable access control.
Mathematics	Average	A rounded division during dASF redemption might lead to overcharging users.
Complexity	Good	Contracts are well-designed and easy to understand.
Libraries	Good	The Solidity contract uses the OpenZeppelin library. No libraries are present in the Vyper implementation.
Decentralization	Average	While contracts are mostly permissionless, governance can enable privileged accounts to redeem dASF for free.
Code stability	Good	The codebase remained stable throughout the review.
Documentation	Average	Although no high-level documentation was provided, the contracts include inline comments describing their intended behavior.
Monitoring	Good	Monitoring events are in place, despite missing logging at initialization.
Testing and verification	Average	The codebase includes a testing suite with several tests that fork mainnet. However, coverage is not comprehensive as there aren't many tests exploring different execution paths, particularly in the price feed and flash dump contracts, which only have one test case each.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
    - Findings that can improve the gas efficiency of the contracts.
- Informational
    - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

None.

Medium Findings

1. Medium - Users may burn extra DASF and pay extra USAD when VeASF rounds down

When redeeming DASF for VeASF, users burn their DASF and pay a certain amount of USAD. Then, the ASF gets locked for a particular duration into the VeASF contract.

However, when locking ASF for VeASF, the contract will round down to the whole token (1e18), meaning that sending 1.5 DASF will only lock 1 ASF. The 0.5 ASF left will be burned, and the user will have paid a little extra USAD for nothing.

Technical Details

The function `redeem()` is in charge of burning DASF, collecting USAD, and locking ASF for VeASF for the users.

When calling that function, users can input any amount of DASF. This amount will be burned from the user's DASF balance and used to compute the USAD payment required.

Then, when locking the ASF for VeASF, the `VEASF.lock()` function takes in `amount // UNIT`, which rounds down to the nearest whole token.

When calling the function, if the user inputs an amount not rounded to the nearest token, he will get that dust burned for nothing and might pay a slightly higher amount of USAD than needed.

Impact

Medium. Users may lose DASF dust and pay a slightly higher USAD amount when redeeming.

Recommendation

Round down the `amount` parameter at the beginning of the function and use it to compute the DASF to burn and USAD to transfer.

Developer Response

Fixed here.

Low Findings

None.

Gas Saving Findings

1. Gas - Price computation repetition inside `latestRoundData()`

Technical Details

Inside `latestRoundData()` the price is computed twice, first when calling `_get_price_with_timestamp()` then again inside the return by calling `_get_price()`.

The first call saves the result inside `price`, this variable is not used.

Impact

Gas.

Recommendation

Remove the call to `_get_price()` and return the `price` variable instead.

Developer Response

Fixed here.

2. Gas - Simplify price feed computation

The ETH price is scaled to match the unit precision only to be downscaled later.

Technical Details

The implementation of `_get_eth_price()` scales the ETH Chainlink price by `1e10` which is later downscaled again in `_calc_price()`.

Impact

Gas savings.

Recommendation

Remove the scaling from `_get_eth_price()`:

```
-   return convert(price * 10**10, uint256), updated_at
+   return convert(price, uint256), updated_at
```

Adjust the downscaling in `_calc_price()`:

```
-   staticcall CURVE_POOL.price_oracle() * eth_price // UNIT,
+   staticcall CURVE_POOL.price_oracle() * eth_price // 10**8,
```

Developer Response

Fixed here. Also renamed `UNIT` to `WAD`.

Informational Findings

1. Informational - The Balancer flashloan fee is ignored

The flashloan fee, currently at zero, is ignored when repaying the loan to Balancer.

Technical Details

flash_dump.vy#L155

Impact

Informational.

Recommendation

Consider adding the fee to the repayment amount so the implementation continues functioning if Balancer enables the fees.

Developer Response

Will use a different provider if they do so.

2. Informational - Missing events for parameters during construction

While setters emit events, parameters defined in constructors don't emit initialization events.

Technical Details

- ownable_2step.vy#L47
- redemption.vy#L105-L109

Impact

Informational.

Recommendation

Consider logging events while initializing the configuration settings in contract constructors.

Developer Response

Fixed here.

3. Informational - Missing validation for payee address in redemption.vy

The `payee` variable is initialized without validating against the empty address.

Technical Details

redemption.vy#L109.

Impact

Informational.

Recommendation

Check that `payee != empty(address)` in the constructor to mimic the setter behavior.

Developer Response

Fixed here.

4. Informational - Missing slippage check for `coin_required`

Technical Details

When calling the function `redeem()`, the user will pay a certain amount of USAD depending on the current price of ASF and the number of weeks to lock them for.

There is currently no way for a user to input the maximum price of USAD he is willing to pay to redeem his DASF. If the price fluctuates or the discount changes while the transaction is being confirmed, a user may pay a slightly higher or lower price than expected.

Impact

Informational.

Recommendation

Add a `max_coin_required` parameter to the function and revert if `coin_required > max_coin_required`.

Protecting the user from paying a higher USAD price than expected.

Developer Response

Fixed here.

Final Remarks

The DASF codebase is simple and well-structured. While using a TWAP oracle may raise concerns, it has been implemented safely, with no significant vulnerabilities identified. The team has demonstrated responsiveness and efficiency in addressing reported issues and applying improvements.