SHERLOCK

Security Review For Aegis

Public Best Efforts Audit Contest Prepared For:	**Aegis**
Lead Security Expert:	**hildingr**
Date Audited:	**April 28 - May 3, 2025**
Final Commit:	**c8c18a7**

Introduction

Aegis.im is a Bitcoin-backed, delta-neutral stablecoin with real-time transparency, built-in funding-rate yield, and zero reliance on the fiat banking system.

This contest focuses on the core YUSD contracts–mint/redeem, rewards distribution, and supporting libraries

Scope

Repository: Aegis-im/aegis-contracts

Audited Commit: eaaf21ec7f3a9bf30a2aadd7118499b7bcf43681

Final Commit: c8c18a744f4a6e4d9fdff1cc6e8a4b3428fe45f6

Files:

- contracts/AegisConfig.sol
- contracts/AegisMinting.sol
- contracts/AegisOracle.sol
- contracts/AegisRewards.sol
- contracts/YUSD.sol
- contracts/lib/ClaimRewardsLib.sol
- contracts/lib/OrderLib.sol

Final Commit Hash

c8c18a744f4a6e4d9fdff1cc6e8a4b3428fe45f6

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
1	2

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0xDLG
0xJoyBoy03
0xNForcer
0xapple
0xaxaxa
0xbakeng
0xc0ffEE
0xeix
0xpiken
0xrex
Asher
Fade
FalseGenius
Ikigai
Ironsidesec
Kvar
OpaBatyo
QuillAudits_Team
Ryonen
aslanbek
asui
bigbear1229
farman1094
gesha17
gkrastenov
harry
hildingr
iamandreiski
irorochadere
itsgreg
mahdifa
mahdikarimi
mgnfy.view
molaratai
moray5554
onthehunt
phoenixv110
ptsanev
s0x0mtee
shealtielanz
silver_eth
t0x1c
turvec
vinica_boy
w33kEd
x0x0
xAlismx

Issue H-1: Insolvency as YUSD will depeg overtime as the redemption fees are disbursed with no collaterals backing them.

Source: https://github.com/sherlock-audit/2025-04-aegis-op-grant-judging/issues/1

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xNForcer, 0xapple, 0xaxaxa, 0xeix, 0xpiken, 0xrex, Asher, Fade, FalseGenius, Kvar, QuillAudits_Team, Ryonen, aslanbek, asui, bigbear1229, farman1094, gesha17, gkrastenov, harry, hildingr, iamandreiski, irorochadere, itsgreg, mahdifa, mgnfy.view, molaratai, moray5554, onthehunt, phoenixv110, s0x0mtee, shealtielanz, silver_eth, t0x1c, turvec, vinica_boy, w33kEd, x0x0

Summary

On the call to `approveRedeemRequest()` the collateral amount which was used initially to get the `yusdAmount` in the `mint()` function, is calculated via the `_calculateRedeemMinCollateralAmount()`.

Here's a run down, please take a good gander.

```
File: AegisMinting.sol

 function approveRedeemRequest(string calldata requestId, uint256 amount) external
↪  nonReentrant onlyRole(FUNDS_MANAGER_ROLE) whenRedeemUnpaused {
    RedeemRequest storage request =
↪  _redeemRequests[keccak256(abi.encode(requestId))];
    if (request.timestamp == 0 || request.status != RedeemRequestStatus.PENDING) {
      revert InvalidRedeemRequest();
    }
    if (amount == 0 || amount > request.order.collateralAmount) {
      revert InvalidAmount();
    }

    uint256 collateralAmount =
↪  _calculateRedeemMinCollateralAmount(request.order.collateralAsset, amount,
↪  request.order.yusdAmount);
//code snip ... some checks.
    uint256 availableAssetFunds =
↪  _untrackedAvailableAssetBalance(request.order.collateralAsset);
    if (availableAssetFunds < collateralAmount) {
      revert NotEnoughFunds();
```

```
    }

    // Take a fee, if it's applicable
    // @audit  fee amount is taken without accounting for the collateral backing
↪   thereby leading to a depegging of the YUSD
    (uint256 burnAmount, uint256 fee) =
↪   _calculateInsuranceFundFeeFromAmount(request.order.yusdAmount, redeemFeeBP);
    if (fee > 0) {
      yusd.safeTransfer(insuranceFundAddress, fee);
    }

    request.status = RedeemRequestStatus.APPROVED;
    totalRedeemLockedYUSD -= request.order.yusdAmount;
   //@audit transfers the entire collateral amount to the user.
    IERC20(request.order.collateralAsset).safeTransfer(request.order.userWallet,
↪   collateralAmount);
    yusd.burn(burnAmount);

    emit ApproveRedeemRequest(requestId, _msgSender(), request.order.userWallet,
↪   request.order.collateralAsset, collateralAmount, burnAmount, fee);
  }
```

The issue is simply, it sends the entire collateral amount backing the given `yusdAmount` to the user, the takes a fee out of the `yusdAmount` and burns the rest, but given fees left aren't backed by any collateral the fee are worthless but would still be redeemable for any of the collateral assets in the contract thereby causing negative yield for the protocol. As fees accumulate, as seen in the code the fees can even rise to 50% it faster the higher the redemption fees are and are accumulated.

Severity Justification.

- While this might at first not seem like a big deal anyone understanding the dynamics of the protocol would see that over time, the stables backing the `YUSD` will be depeleted causing the price of `YUSD` to crash make the contracts insolvent.

Mitigation

The collateral amount to be sent back to the user should be the equivalence in value of the the `yusd` to be burnt on the call to `approveRedeemRequest()` The code snippet should be changed as follows.

```
function approveRedeemRequest(string calldata requestId, uint256 amount) external
↪   nonReentrant onlyRole(FUNDS_MANAGER_ROLE) whenRedeemUnpaused {
    RedeemRequest storage request =
↪   _redeemRequests[keccak256(abi.encode(requestId))];
    if (request.timestamp == 0 || request.status != RedeemRequestStatus.PENDING) {
```

```
     revert InvalidRedeemRequest();
   }
   if (amount == 0 || amount > request.order.collateralAmount) {
     revert InvalidAmount();
   }
++   // Take a fee, if it's applicable
++  (uint256 burnAmount, uint256 fee) =
↪  _calculateInsuranceFundFeeFromAmount(request.order.yusdAmount, redeemFeeBP);
++  if (fee > 0) {
++  yusd.safeTransfer(insuranceFundAddress, fee);
++    }

--  uint256 collateralAmount =
↪  _calculateRedeemMinCollateralAmount(request.order.collateralAsset, amount,
↪  request.order.yusdAmount);
++  uint256 collateralAmount =
↪  _calculateRedeemMinCollateralAmount(request.order.collateralAsset, amount,
↪  burnAmount);
   /*
    * Reject if:
    * - asset is no longer supported
    * - smallest amount is less than order minAmount
    * - order expired
    */
   if (
     !_supportedAssets.contains(request.order.collateralAsset) ||
     collateralAmount < request.order.slippageAdjustedAmount ||
     request.order.expiry < block.timestamp
   ) {
     _rejectRedeemRequest(requestId, request);
     return;
   }

   uint256 availableAssetFunds =
↪  _untrackedAvailableAssetBalance(request.order.collateralAsset);
   if (availableAssetFunds < collateralAmount) {
     revert NotEnoughFunds();
   }

--  // Take a fee, if it's applicable
--  (uint256 burnAmount, uint256 fee) =
↪  _calculateInsuranceFundFeeFromAmount(request.order.yusdAmount, redeemFeeBP);
--  if (fee > 0) {
--    yusd.safeTransfer(insuranceFundAddress, fee);
--  }

   request.status = RedeemRequestStatus.APPROVED;
   totalRedeemLockedYUSD -= request.order.yusdAmount;
```

```
    IERC20(request.order.collateralAsset).safeTransfer(request.order.userWallet,
↪    collateralAmount);
    yusd.burn(burnAmount);

    emit ApproveRedeemRequest(requestId, _msgSender(), request.order.userWallet,
↪    request.order.collateralAsset, collateralAmount, burnAmount, fee);
  }
```

Issue M-1: Collateral can get stuck in the minting contract under certain conditions or be accounted as profit

Source: https://github.com/sherlock-audit/2025-04-aegis-op-grant-judging/issues/77

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

iamandreiski, vinica_boy

Summary

The predominant narrative of Aegis is that they aren't holding any funds in their minting contract. This means that all collateral which was transferred to the contract for the purpose of minting YUSD is transferred to their custodial partners for safekeeping OR is accounted for as profit which would result in YUSD being minted in exchange for it, and then said collateral transferred to their custodial partners for safekeeping.

This warrants the need for a 2-step redeem flow which first registers the redeem request, and then withdraws the needed collateral from the custodial partners after which the redeem can be accepted and the collateral transferred to the user. The problem with this approach is that these funds are ”untracked”, together with the profit.

There are multiple ways in which collateral can get ”stuck” in the contract OR be mistaken for profit:

- A redeem request was rejected or cancelled;
- The redeem request acceptance reverts due to any number of reasons such as the user was placed on a blacklist and/or the order has expired;
- A `depositIncome` call mistakenly accounts untracked collateral funds as profit, resulting in the minting of non-collateralized YUSD which contributes to its de-pegging.

Root Cause

The current design of the system doesn't account (virtually) for both the funds which are to-be-redeemed, as well as the income (i.e. profit). The problem of not accounting for the redeem funds is that they can either be mistaken for an income, or remain stuck in the contract with no way to return them to the custodial partners.

When a user requests a redeem, the request is placed and pending as part of `_redeemRequests`:

```
_redeemRequests[keccak256(abi.encode(requestId))] =
↪  RedeemRequest(RedeemRequestStatus.PENDING, order, block.timestamp);
```

Once the "untracked" funds become available in the minting contract (i.e. are withdrawn from the custodial partners), the redeem request can be accepted/approved:

```
  uint256 availableAssetFunds =
↪  _untrackedAvailableAssetBalance(request.order.collateralAsset);
  if (availableAssetFunds < collateralAmount) {
    revert NotEnoughFunds();
  }


function _untrackedAvailableAssetBalance(address _asset) internal view returns
↪  (uint256) {
  uint256 balance = IERC20(_asset).balanceOf(address(this));
  if (balance < _custodyTransferrableAssetFunds[_asset] + assetFrozenFunds[_asset])
↪  {
    return 0;
  }


  return balance - _custodyTransferrableAssetFunds[_asset] -
↪  assetFrozenFunds[_asset];
}
```

The problem with this approach is that redeem requests are expected to fail / get cancelled after the funds were withdrawn from the custodial partners to the minting contract.

- User could cancel their redeem requests after the funds were already withdrawn to the contract, but before the request is approved;
- An exchange ratio change has prompted the call to be rejected due to slippage;
- The user was placed on a blacklist and the call reverts;
- The order has expired;

Any of (but not limited to) the above-mentioned scenarios could result in the user not being able to claim their collateral, the request being rejected/cancelled and the funds remaining stuck in the minting contract. Since currently, there's no way to send untracked funds to the custodial partners, they will remain stuck within the minting contract.

This is also a problem if the redeem request was a very large amount which affects the balancing strategy of Aegis, and not being able to utilize those funds as part of their delta-neutral position strategy.

A second point of this report is that if the funds stuck in the contract are ”mistaken” or otherwise intentionally declared as profit, this would lead to the minting of undercollateralized YUSD.

Considering that the ”extra” collateral in the minting contract can be declared as profit by minting adequate and corresponding YUSD amounts through `depositIncome`, the function assumes that the untracked collateral in-question wasn't utilized for the minting of YUSD tokens:

https://github.com/sherlock-audit/2025-04-aegis-op-grant/blob/main/aegis-contracts/contracts/AegisMinting.sol#L407-L424

And that's why it's being treated as a normal mint flow in which the user provides collateral and receives a corresponding amount of YUSD tokens in exchange for the collateral supplied.

Internal Pre-conditions

1. A certain collateral amount is deposited to the minting contract from the custodial partners for the purpose of approving redeem requests;
2. Said redeem requests can't be approved due to a multitude of reasons and are rejected or cancelled, leading to the already withdrawn ”untracked” collateral to be stuck in the contract and unable to be sent back to the custodial counterparts.

External Pre-conditions

N/A

Attack Path

N/A

Impact

The untracked collateral amounts withdrawn to the minting contract which are unused (i.e. haven't been utilized for redeems) will remain stuck in the contract or can be declared as profit which will lead to the minting of undercollateralized YUSD.

PoC

No response

Mitigation

Include an admin controlled mechanism to be able to repurpose unused redeem funds as `_custodyTransferrableAssetFunds` or make a separate request funds flow which will track the amounts and enable redeem approvals (and if they get rejected, the funds get atomically repurposed).

Issue M-2: A whale adversary can grief the redeem functionality through redeem limit consumption

Source: https://github.com/sherlock-audit/2025-04-aegis-op-grant-judging/issues/497

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xDLG, 0xJoyBoy03, 0xNForcer, 0xbakeng, 0xc0ffEE, Ikigai, Ironsidesec, OpaBatyo, farman1094, gkrastenov, hildingr, iamandreiski, itsgreg, mahdikarimi, phoenixv110, ptsanev, vinica_boy, xAlismx

Summary

The failure to not account redemptions into the limit after the transaction has been approved can lead to griefing attempts by users with sizeable YUSD token balance meanwhile the redemption can still expire.

Root Cause

In AegisMinting.sol:785-802, the _checkMintRedeemLimit function increments limits.currentPeriodTotalAmount when a user creates a redeem request, but neither the withdrawRedeemRequest function (lines 373-390) nor the _rejectRedeemRequest function (lines 702-711) decrements this counter when YUSD is returned to the user.

```
function _checkMintRedeemLimit(MintRedeemLimit storage limits, uint256 yusdAmount)
 ↪  internal {
    // ... validation checks ...
    limits.currentPeriodTotalAmount += yusdAmount; // Incremented but never
 ↪  decremented on withdraw/reject
}
```

Internal Pre-conditions

N/A

External Pre-conditions

1. Adversary needs to have enough YUSD tokens to make redeem requests that approach redeemLimit.maxPeriodAmount.

Attack Path

1. Adversary identifies the current redeemLimit.maxPeriodAmount and redeemLimit.periodDuration.
2. Adversary creates multiple valid Order structs for OrderType.REDEEM with different nonces and additionalData (to create unique request IDs).
3. Attacker obtains valid signatures for these orders from the trustedSigner.
4. Attacker calls requestRedeem with these orders multiple times, locking YUSD each time.
5. Attacker continues step 4 until redeemLimit.currentPeriodTotalAmount approaches redeemLimit.maxPeriodAmount.
6. The limit available for redeem is now almost exhausted for the current period.
7. Any user (including legitimate users) calling requestRedeem for more than the small remaining limit will have their transaction revert with LimitReached error.
8. After order.expiry has passed, the attacker calls withdrawRedeemRequest for each request.
9. The attacker receives back all their locked YUSD, but redeemLimit.currentPeriodTotalAmount remains unchanged.
10. The DoS persists until the current period ends and redeemLimit resets naturally.

Impact

Legitimate users cannot execute the core functionality of requesting a redeem (requestRedeem) for the duration of the limit period under attack. This DoS prevents users from exiting their positions when desired, which could lead to financial loss if users need to redeem during specific market conditions. The attacker only temporarily locks their funds and loses nothing but gas fees in the process.

PoC

Check Attack Path

Mitigation

Adjust for the limit after the redemption has actually been approved to ensure whales have fully commited with redeeming.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.