SHERLOCK

Security Review For Nerite

Collaborative Audit Prepared For:	Nerite
Lead Security Expert(s):	0x73696d616f
	GalloDaSballo
Date Audited:	February 3 - February 17, 2025
Final Commit:	5541c61

Introduction

Nerite is a stablecoin CDP protocol based on Liquity V2. It adds new functionality like more collateral types, delegation of collateral tokens, and most notably a native superfluid stablecoin which is streamable.

Scope

Repository: NeriteOrg/nerite

Audited Commit: b957c0417697ac4ee9c4aa6d4753739817f169df

Final Commit: 5541c61ee4e78991fa0f7e29eebff5e6680735bd

Files:

- contracts/src/ActivePool.sol
- contracts/src/AddressesRegistry.sol
- contracts/src/BoldToken.sol
- contracts/src/BorrowerOperations.sol
- contracts/src/CollSurplusPool.sol
- contracts/src/CollateralRegistry.sol
- contracts/src/DefaultPool.sol
- contracts/src/Dependencies/AddRemoveManagers.sol
- contracts/src/Dependencies/AggregatorV3Interface.sol
- contracts/src/Dependencies/Constants.sol
- contracts/src/Dependencies/IOsTokenVaultController.sol
- contracts/src/Dependencies/IStaderOracle.sol
- contracts/src/Dependencies/LiquityBase.sol
- contracts/src/Dependencies/LiquityMath.sol
- contracts/src/Dependencies/Ownable.sol
- contracts/src/GasPool.sol
- contracts/src/HintHelpers.sol
- contracts/src/Interfaces/IActivePool.sol
- contracts/src/Interfaces/IAddRemoveManagers.sol
- contracts/src/Interfaces/IAddressesRegistry.sol
- contracts/src/Interfaces/IBoldRewardsReceiver.sol
- contracts/src/Interfaces/IBoldToken.sol

- contracts/src/Interfaces/IBorrowerOperations.sol
- contracts/src/Interfaces/ICollSurplusPool.sol
- contracts/src/Interfaces/ICollateralRegistry.sol
- contracts/src/Interfaces/ICommunityIssuance.sol
- contracts/src/Interfaces/IDefaultPool.sol
- contracts/src/Interfaces/IHintHelpers.sol
- contracts/src/Interfaces/IInterestRouter.sol
- contracts/src/Interfaces/ILQTYStaking.sol
- contracts/src/Interfaces/ILQTYToken.sol
- contracts/src/Interfaces/ILiquityBase.sol
- contracts/src/Interfaces/IMainnetPriceFeed.sol
- contracts/src/Interfaces/IMultiTroveGetter.sol
- contracts/src/Interfaces/IPriceFeed.sol
- contracts/src/Interfaces/IRETHPriceFeed.sol
- contracts/src/Interfaces/IRETHToken.sol
- contracts/src/Interfaces/ISortedTroves.sol
- contracts/src/Interfaces/IStabilityPool.sol
- contracts/src/Interfaces/IStabilityPoolEvents.sol
- contracts/src/Interfaces/ITreeETHPriceFeed.sol
- contracts/src/Interfaces/ITreeETHToken.sol
- contracts/src/Interfaces/ITroveEvents.sol
- contracts/src/Interfaces/ITroveManager.sol
- contracts/src/Interfaces/ITroveNFT.sol
- contracts/src/Interfaces/IWETH.sol
- contracts/src/Interfaces/IWSTETH.sol
- contracts/src/Interfaces/IWSTETHPriceFeed.sol
- contracts/src/Interfaces/IWeETHPriceFeed.sol
- contracts/src/Interfaces/IWeETHToken.sol
- contracts/src/MultiTroveGetter.sol
- contracts/src/NFTMetadata/MetadataNFT.sol
- contracts/src/PriceFeeds/ARBPriceFeed.sol

- contracts/src/PriceFeeds/COMPPriceFeed.sol
- contracts/src/PriceFeeds/CompositePriceFeed.sol
- contracts/src/PriceFeeds/MainnetPriceFeedBase.sol
- contracts/src/PriceFeeds/RETHPriceFeed.sol
- contracts/src/PriceFeeds/TokenPriceFeedBase.sol
- contracts/src/PriceFeeds/UNIPriceFeed.sol
- contracts/src/PriceFeeds/WETHPriceFeed.sol
- contracts/src/PriceFeeds/WSTETHPriceFeed.sol
- contracts/src/PriceFeeds/WeETHPriceFeed.sol
- contracts/src/PriceFeeds/XVSPriceFeed.sol
- contracts/src/PriceFeeds/tBTCPriceFeed.sol
- contracts/src/PriceFeeds/treeETHPriceFeed.sol
- contracts/src/SortedTroves.sol
- contracts/src/StabilityPool.sol
- contracts/src/TroveManager.sol
- contracts/src/TroveNFT.sol
- contracts/src/Types/BatchId.sol
- contracts/src/Types/LatestBatchData.sol
- contracts/src/Types/LatestTroveData.sol
- contracts/src/Types/TroveChange.sol
- contracts/src/Types/TroveId.sol
- contracts/src/Zappers/BaseZapper.sol
- contracts/src/Zappers/GasCompZapper.sol
- contracts/src/Zappers/LeftoversSweep.sol
- contracts/src/Zappers/LeverageLSTZapper.sol
- contracts/src/Zappers/LeverageWETHZapper.sol
- contracts/src/Zappers/WETHZapper.sol

Final Commit Hash

5541c61ee4e78991fa0f7e29eebff5e6680735bd

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.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	8	15

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Adjusting of healthy positions could be dosses by the _moveTokensFromAdjustment

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/81

Summary

`_moveTokensFromAdjustment` checks that the current debt is above the limit even when an operation is reducing debt, which can lead to DOS and in some cases can be weaponized to prevent repaying

Vulnerability Detail

`_moveTokensFromAdjustment` checks if the trove adjustment will move the `entireSystemDebt` to be above the limit

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/BorrowerOperations.sol#L1218-L1233

```
// This function mints the BOLD corresponding to the borrower's chosen debt increase
// (it does not mint the accrued interest).
function _moveTokensFromAdjustment(
    address withdrawalReceiver,
    TroveChange memory _troveChange,
    IBoldToken _boldToken,
    IActivePool _activePool
) internal {
    require(troveManager.getDebtLimit() >= troveManager.getEntireSystemDebt() +
 ↪  _troveChange.debtIncrease, "BorrowerOperations: Debt limit exceeded.");

    if (_troveChange.debtIncrease > 0) {
        _boldToken.mint(withdrawalReceiver, _troveChange.debtIncrease);
    } else if (_troveChange.debtDecrease > 0) {
        _boldToken.burn(msg.sender, _troveChange.debtDecrease);
    }
```

But the check is ignoring the scenario in which we're already above the limit

Because of the fact that checks are only present in `OpenTrove`, `withdrawBold` and `adjustTrove` then if we perform a few adjustments on Batches that have a lot of debt (which will not revert since there's no check for being above the limit, and such a check shouldn't be there either way)

Impact

Then adjusting a small position may be DOSed (even repaying) once the debt is above the cap

This could be used to forcefully cause a position that is trying to repay debt to get liquidated

Tool Used

Manual Review

Recommendation

The check for being above the limit should be performed only for operations that are increasing the debt, and not for operations in which the debt is being repaid

Discussion

cupOJoseph

fix

https://github.com/NeriteOrg/nerite/pull/46

cupOJoseph

reviewed after redeploy.

Issue M-2: Multiplicative factor on debt limit will cause issues once limits are reduced

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/82

Summary

Vulnerability Detail

The function `updateDebtLimit` is as follows:

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/CollateralRegistry.sol#L310-L319

```
// Update the debt limit for a specific TroveManager
function updateDebtLimit(uint256 _indexTroveManager, uint256 _newDebtLimit)
↪  external onlyGovernor {
    //limited to increasing by 2x at a time, maximum. Decrease by any amount.
    uint256 currentDebtLimit = getTroveManager(_indexTroveManager).getDebtLimit();
    if (_newDebtLimit > currentDebtLimit) {
        require(_newDebtLimit <= currentDebtLimit * 2, "CollateralRegistry: Debt
↪  limit increase by more than 2x is not allowed");
    }
    getTroveManager(_indexTroveManager).setDebtLimit(_newDebtLimit);
}
```

Due to the multiplicative nature of limits, setting to 0 will prevent raising them

Setting to 1 will raise to 2

You'd need to set some bound at which the limit goes back without using this multiplicative logic

However even that could be somewhat off

Overall Multiplicative Limits are not great

This means that, for a starting limit that is set in a sound way, such as 2e24 then the math is fairly sound

However, anyone could bribe governance to change the caps once to let's say 0 or 1 wei

For 0 weis then the cap will be permanently broken and the branch will never be usable again

For 1 wei, the cap will take a very long time before any meaningful amount will be borrowable

Impact

In lack of boundaries, one vote that massively reduces caps will disable borrowing for a considerable amount of times

Code Snippet

Tool Used

Manual Review

Recommendation

Consider having some sort of a ”Enable vs Disable” threshold (e.g. 2 Million USDN)

And from there you should be safe to apply a multiplicative cap

Although generally speaking multiplicative operations tend to be more error prone as they are counter intuitive

It may be best to separate assets by tiers and based on the tiers allow certain changes over time

Or perhaps allow changes at different intervals with different impacts

Discussion

cupOJoseph

https://github.com/NeriteOrg/nerite/pull/43

Fixed by setting up to the initially set limit.

cupOJoseph

reviewed after redeploy.

Issue M-3: Some oracle pairs will require raising the Base Redemption Fee

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/89

Summary

Due to composite deviation thresholds, the average deviation threshold of some assets will go above 50 BPS

To 55 BPS in the case of xETH / ETH * ETH / USD

This can leak a moderate amount of value through redemptions

Vulnerability Detail

Preemptive research was done by Recon to gather available Oracle Configurations

The whole research is available here: https://gist.github.com/Simon-Busch/9b4ac3669974499aa6afdb656408cf2a

The currently available combinations for xETH/ETH can leak around 5 BPS of value before the oracle will trigger an update

The delay between an update being triggered and it resulting in a new price can further leak value that is hard to quantify

Impact

Value leak through Redemptions

Tool Used

Manual Review

Recommendation

It may be best to raise the redemption base fee to 55 BPS (just 5 BPS extra) Or to negotiate more accurate oracles with providers

Discussion

cupOJoseph

yeah this is a very very good find.

I've asked for some help from Liquity team with this as I'm not sure this can be set per collateral type. I will update soon.

cupOJoseph

It is not possible to update this per collateral so the entire system will be updated to 55BP redemption fee. This will effect the peg

https://github.com/NeriteOrg/nerite/pull/65

cupOJoseph

updated REDEMPTION_FEE_FLOOR in constants.sol to 55 BPs^

reviewed.

Issue M-4: tBTC should protect against redemption value skim by using the BTC/USD feed

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/90

Summary

tBTC is a tokenized BTC deposit on the Threshold System

However it's a separate token that can be redeemed for BTC

Due to this tBTC can trade above and below peg with BTC

This can create opportunities for arbitrage by redeeming Bold -> tBTC, when tBTC is underpriced

Vulnerability Detail

Oracle Drift is the difference between the Oracle Reported Price and the Market Price of a token

Because Oracles have a Deviation Threshold and require some time to update, then their values can be inaccurate, creating for arbitrage opportunities

In the case of Nerite and tBTC the arbitrage would work as follows:

- tBTC is trading at parity with USDN ($1 of tBTC = 1 USDN)
- The oracle is reporting that tBTC is trading below peg
- An arbitrageur can redeem tBTC as if it was worth less than $1 and receive $1 * 1/depeg value in ETH

This causes the system to leak value as redemptions would happen not because BOLD depegged, but because the Oracle is under-reporting the value of tBTC

Impact

Value leak through Redemptions

You can over-price by using BTC/USD so that redemptions are properly priced until tBTC breaks peg / is actually exploited

For example you could have a 1 or 2% threshold for redemptions during which you use the max between BTC/USD and tBTC/USD prices

And once the price is too different you'd default back to tBTC/USD

Tool Used

Manual Review

Recommendation

Implement logic similar to that of wstETH/USD to protect against Redemption Skimming

Discussion

cupOJoseph

https://github.com/NeriteOrg/nerite/pull/49 fixed here

cupOJoseph

reviewed

Issue M-5: BoldToken initialization can be frontrun to silently mint/approve bold to an attacker

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/94

Summary

`BoldToken::initialize()` is frontrunnable.

Vulnerability Detail

`BoldToken::initialize()` is called seperately after the constructor, which means that if sent in different txs (depending on deployment scripts, it may be possible to mitigate if sent via a multicall), an attacker can frontrun the deployment, initialize the `BoldToken` to a malicious implementation, mint/approve `Bold` to their address and then set the implementation to `address(0)` so the script does not revert and the `BoldToken::initialize()` call succeeds.

Impact

Attackers mints/gets `Bold` for free.

Code Snippet

https://github.com/sherlock-audit/2025-02-nerite/blob/main/nerite/contracts/src/BoldToken.sol#L59-L72 https://github.com/superfluid-finance/protocol-monorepo/blob/1edc4ed3ddafa87d42c1155e870532e2a5a80470/packages/ethereum-contracts/contracts/upgradability/UUPSProxy.sol#L25-L29

Tool Used

Manual Review

Recommendation

Use a multicall to deploy or move the `BoldToken::initialize()` logic to the constructor.

Discussion

cupOJoseph

BoldToken::initialize() cannot be in the constructor for reasons related to superfluid protocol. The contract storage slots cannot be called because they dont exist yet.

This will be solved by formally verifying after deployment.

Issue M-6: tETH Rate Provider is Missing / provided one is incompatible with the oracles in scope

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/100

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

Summary

The Rate Provider in scope for tETH is pricing tETH against wstETH: https://arbiscan.io/address/0x98a977Ba31C72aeF2e15B950Eb5Ae3158863D856

This makes it impossible to use any oracle in scope

Recommendation

Given other assessment, consider dropping tETH

Discussion

cupOJoseph

Acknowledged. This is addressed by multiplying by the wsteth/USD price and getting a USD price of tETH. Although this creates a higher level of uncertainty and level of error, those are addressed by more strict risk params for the collateral and a very conservative debt limit.

Issue M-7: stEthPerToken is not available on arbitrum and should be replaced by getRate

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/101

Summary

Since wstETH is a bridged token the function `stEthPerToken` is unavailable on Arbitrum

Vulnerability Detail

This call will fail: https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/PriceFeeds/WSTETHPriceFeed.sol#L71-L79

```
function _getCanonicalRate() internal view override returns (uint256, bool) {
    uint256 gasBefore = gasleft();

    try IWSTETH(rateProviderAddress).stEthPerToken() returns (uint256
↪   stEthPerWstEth) {
        // If rate is 0, return true
        if (stEthPerWstEth == 0) return (0, true);

        return (stEthPerWstEth, false);
    } catch {
```

Since the wstETH token in arbitrum is a generic bridged ERC20: https://arbiscan.io/address/0x0fBcbaEA96Ce0cF7Ee00A8c19c3ab6f5Dc8E1921#readContract

Impact

The oracle is unusable

Code Snippet

Tool Used

Manual Review

Recommendation

Change the call to a rate provider:

`provider.getRate()`

https://arbiscan.io/address/0xf7c5c26b574063e7b098ed74fad6779e65e3f836#readContract

Which uses https://arbiscan.io/address/0xB1552C5e96B312d0Bf8b554186F846C40614a540#readContract

Source: https://balancer.fi/pools/arbitrum/v2/0xb61371ab661b1acec81c699854d2f911070c059e000000000000000000000516

Discussion

cupOJoseph

Great find, we're ready for arbitrum!

Fixed here: https://github.com/NeriteOrg/nerite/pull/56

cupOJoseph

reviewed

Issue M-8: tETH poses existential threat to Nerite

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/102

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

Summary

tETH has a owner that is a multisig

This means that if the multisig turns rogue the signers will be able to steal all value (until borrow caps) from Nerite

Vulnerability Detail

tETH Doesn't seem to meet any reasonable criteria for safety:

- Uses wstETH Rate internally -> Most likely has coding issues
- Owner can change rate provider at any time
- Owner is a 5/7 multisig

The owner can rug the token at will at any time

I don't believe the setup is safe and the setup is incompatible with a slow, governance minimized system

Their safe: https://arbiscan.io/address/0xB35A37f1614ca61379526B5a1080D853517B5824#readProxyContract

Impact

Code Snippet

Code that can be used to rug pull Nerite:

```
/// @notice Mints new tokens for a given address.
/// @param account The address to mint the new tokens to.
/// @param amount The number of tokens to be minted.
/// @dev this function increases the total supply.
function mint(address account, uint amount) external onlyMinter {
  _mint(account, amount);
}

/// @notice Burns tokens from the sender.
/// @param amount The number of tokens to be burned.
/// @dev this function decreases the total supply.
function burn(uint amount) external onlyBurner {
  _burn(_msgSender(), amount);
}

/// @notice Burns tokens from a given address.
/// @param account The address to burn tokens from.
/// @param amount The number of tokens to be burned.
/// @dev this function decreases the total supply.
function burnFrom(address account, uint amount) external onlyBurner {
  _spendAllowance(account, _msgSender(), amount);
  _burn(account, amount);
}

// ================================================================
// |                            Roles                             |
// ================================================================

/// @notice grants both mint and burn roles to `burnAndMinter`.
/// @dev calls public functions so this function does not require
/// access controls. This is handled in the inner functions.
function grantMintAndBurnRoles(address burnAndMinter) external {
  grantMintRole(burnAndMinter);
  grantBurnRole(burnAndMinter);
}

/// @notice Grants mint role to the given address.
/// @dev only the owner can call this function.
function grantMintRole(address minter) public onlyOwner {
  if (s_minters.add(minter)) {
    emit MintAccessGranted(minter);
  }
}
```

Tool Used

Manual Review

Recommendation

If you really want the tETH team to work with you, they should setup this governance setup I explain in detail: https://x.com/getreconxyz/status/1885249716386226572

Most likely with a 14 days timelock on those functions which pose an existential threat to Nerite

Discussion

cupOJoseph

Acknowledged. This will be addressed by a very low until debt limit on tETH until they add a timelock to their multisig in the coming months, and has been acknowledged by the treehouse team. However, no code changes to Nerite are needed in our opinion.

Issue L-1: A bunch of checks with TODO that should be removed are left in the codebase

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/83

Summary

The codebase has some TODOs about removing assertions

Vulnerability Detail

The Liquity codebase has been deployed with a few inlined assertions left in their source code

Impact

This causes no impact beside some small amount of gas being lost

Code Snippet

Tool Used

Manual Review

Recommendation

Either delete the TODOs or acknowledge this

Discussion

cupOJoseph

fixed https://github.com/NeriteOrg/nerite/pull/47

cupOJoseph

reviewed after redeploy.

Issue L-2: withdrawBold troveManagerCached.getDebtLimit() >= troveManagerCached.getEntireSystemDebt() + _boldAmount check is unnecessary

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/86

Summary

The function `withdrawBold` has a check to ensure that borrowing won't surpass the `debtLimit` which is unnecesary

Vulnerability Detail

`_moveTokensFromAdjustment` is called by `_adjustTrove` and has the following check

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/BorrowerOperations.sol#L1221-L1228

```
function _moveTokensFromAdjustment(
    address withdrawalReceiver,
    TroveChange memory _troveChange,
    IBoldToken _boldToken,
    IActivePool _activePool
) internal {
    require(troveManager.getDebtLimit() >= troveManager.getEntireSystemDebt() +
↪   _troveChange.debtIncrease, "BorrowerOperations: Debt limit exceeded.");
```

Which makes the check in `withdrawBold` unnecessary

Impact

The check is unnecessary

Tool Used

Manual Review

Recommendation

Remove the check

Discussion

cupOJoseph

removed check https://github.com/NeriteOrg/nerite/pull/51

cupOJoseph

reviewed after deploy.

Issue L-3: getDebtLimit check in OpenTrove is ignoring the additional debt that comes from the adjustment

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/87

Summary

The check for `getDebtLimit` in `openTrove` is ignoring the additional debt paid on the upfront fee

Which is inconsistent with the checks done in `_adjustTrove`

Vulnerability Detail

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/BorrowerOperations.sol#L314-L315

```
require(troveManager.getDebtLimit() >= troveManager.getEntireSystemDebt() +
↪  _boldAmount, "BorrowerOperations: Debt limit exceeded.");
```

The code will later compute:

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/BorrowerOperations.sol#L333-L334

```
vars.entireDebt = _change.debtIncrease + _change.upfrontFee;
_requireAtLeastMinDebt(vars.entireDebt);
```

Which is the actual total amount

Impact

This causes the ability to borrow a bit more than intended

Code Snippet

Tool Used

Manual Review

Recommendation

Acknowledge this or fix the check by moving it later in the code (after the upfrontFee) is computed

Discussion

cupOJoseph

good spot, I feel silly I missed this one. I might actually make this medium

fixed here: https://github.com/NeriteOrg/nerite/pull/52

cupOJoseph

reviewed after redeploy.

Issue L-4: Lack of checks for when the sequencer is down

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/88

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

Summary

The oracles are not checking for when the sequencer is down

Vulnerability Detail

When the Arbitrum Sequencer is down, transactions can go through 2 possible flows:

1) They are DOSed as users are unable to interact with the L2
2) They are force included, after a 24hr delay, txs can be force included ensuring they will get executed

This creates the following undefined states that are better explored now explicitly:

- Arbitrum having a downtime can cause the entire system to shutdown, which will also leak value through urgent redemptions

This happening while the sequencer is down can make it impossible for most people to close their positions as a reaction to the downtime

- Oracles can be wildly out of synch while the sequencer is down

Many oracles on Arbitrum offer high precision (5BPS) however, during a sequencer downtime this could change based on the provider

Impact

These are very low likelihood scenarios, that can leak a extensive amount of value

It's worth noting that the more successful the protocol is the more 0 days agains the Arbitrum Sequencer may be used to extract value

A specific risk is left open by a lack of checking if the sequencer is up:

A threat actor now has an economic bounty of 1% of all AP TVL as if they are able to cause a long enough downtime, they will be able to claim a 1% premium on all assets (using the stale price, which could further inflate the profitability of the attack)

On a $1 Billion TVLurgent redemptions will leak 1%, that’s a 10 million dollar incentive to DOS Arbitrum and shut down Nerite

This is very low likelihood but it does create this incentive for those that research exploits against the Arbitrum sequencer

Tool Used

Manual Review

Recommendation

Document these states and decide whether:

- The system should shutdown and end users should take this risk
- The system should wait for the sequencer to be back up, by using a L2 Sequencer Oracle on top of the other oracles

Discussion

cupOJoseph

Added to our risk disclosures. Users must accept this risk of using L2 networks.

https://docs.nerite.org/docs/technical-documentation/risks

Issue L-5: Base Rate Decay will be slower than intended - Chaduke

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/91

Description:

Note

This is an already known finding from ETHOS (Liquity Fork)

Impact

`_calcDecayedBaseRate` calls `_minutesPassedSinceLastFeeOp` which rounds down by up to 1 minute - 1

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/CollateralRegistry.sol#L222-L227

```
function _calcDecayedBaseRate() internal view returns (uint256) {
    uint256 minutesPassed = _minutesPassedSinceLastFeeOp();
    uint256 decayFactor = LiquityMath._decPow(REDEMPTION_MINUTE_DECAY_FACTOR,
↪   minutesPassed);

    return baseRate * decayFactor / DECIMAL_PRECISION;
}
```

This, in conjunction with the logic `_updateLastFeeOpTime`

https://github.com/ebisufinance/ebisu-money/blob/64ed19640c5f5c0d57a615234eb2ef18ea0a805f/contracts/src/CollateralRegistry.sol#L184-L201

~~~
https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/c⌋
↪   ontracts/src/CollateralRegistry.sol#L172-L180

```solidity
    // Update the last fee operation time only if time passed >= decay interval.
↪   This prevents base rate griefing.
    function _updateLastFeeOpTime() internal {
        uint256 timePassed = block.timestamp - lastFeeOperationTime;

        if (timePassed >= ONE_MINUTE) {
            lastFeeOperationTime = block.timestamp;
            emit LastFeeOpTimeUpdated(block.timestamp);
        }
    }
~~~

```
    function _minutesPassedSinceLastFeeOp() internal view returns (uint256) {
        return (block.timestamp - lastFeeOperationTime) / ONE_MINUTE;
    }

    // Updates the `baseRate` state with math from
↪   `_getUpdatedBaseRateFromRedemption`
    function _updateBaseRateAndGetRedemptionRate(uint256 _boldAmount, uint256
↪   _totalBoldSupplyAtStart) internal {
        uint256 newBaseRate = _getUpdatedBaseRateFromRedemption(_boldAmount,
↪   _totalBoldSupplyAtStart);
```

Will make the decay factor decay slower than intended

Recommendation:

This finding was found in the ETHOS contest by Chaduke:
https://github.com/code-423n4/2023-02-ethos-findings/issues/33

Client:

Zenith:

Discussion

lpetroulakis

Recommended solution merged successfully and tested.

Issue L-6: New debt from adjusted trove is double counted towards the limit

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/93

Summary

When adjusting a trove, the newly minted debt is counted twice, as `troveManager.getEntireSystemDebt()` already includes the new debt.

Vulnerability Detail

`BorrowerOperations::_moveTokensFromAdjustment()` checks if the debt has exceeded the limit by requiring that `troveManager.getDebtLimit() >= troveManager.getEntireSystemDebt() + _troveChange.debtIncrease`. However, when this check is performed, `troveManager.getEntireSystemDebt()` already includes the `_troveChange.debtIncrease` component, as it was added previously in the `vars.activePool.mintAggInterestAndAccountForTroveChange(_troveChange, batchManager);` call.

Impact

Adjusting troves will revert due to the debt limit when they should not.

Code Snippet

https://github.com/sherlock-audit/2025-02-nerite/blob/main/nerite/contracts/src/BorrowerOperations.sol#L670 https://github.com/sherlock-audit/2025-02-nerite/blob/main/nerite/contracts/src/BorrowerOperations.sol#L1227

https://github.com/sherlock-audit/2025-02-nerite/blob/main/nerite/contracts/src/ActivePool.sol#L164 https://github.com/sherlock-audit/2025-02-nerite/blob/main/nerite/contracts/src/ActivePool.sol#L233-L238

Tool Used

Manual Review

Recommendation

`require(troveManager.getDebtLimit() >= troveManager.getEntireSystemDebt(), "BorrowerOperations: Debt limit exceeded.");` should be enough.

Discussion

cupOJoseph

I would mark this as Low since its inaccurate but can only OVER estimate the debt.

Good spot though! Fixed here: https://github.com/NeriteOrg/nerite/pull/53

cupOJoseph

reviewed after redeploy

Issue L-7: Liquidations can fail on a ”revert on 0 transfer” token

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/95

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

Summary

`activePool.sendColl` can be called in reverting ways in the Stability Pool for tokens that revert on zero transfer

Vulnerability Detail

The SP will use the function `_moveOffsetCollAndDebt` which looks as follows:

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/StabilityPool.sol#L572-L587

```
function _moveOffsetCollAndDebt(uint256 _collToAdd, uint256 _debtToOffset) internal
 ↪  {
    // Cancel the liquidated Bold debt with the Bold in the stability pool
    _updateTotalBoldDeposits(0, _debtToOffset);

    // Burn the debt that was successfully offset
    boldToken.burn(address(this), _debtToOffset);

    // Update internal Coll balance tracker
    uint256 newCollBalance = collBalance + _collToAdd;
    collBalance = newCollBalance;

    // Pull Coll from Active Pool
    activePool.sendColl(address(this), _collToAdd);

    emit StabilityPoolCollBalanceUpdated(newCollBalance);
}
```

This will call `activePool.sendColl` even with a 0 value

A revert in `sendColl` can DOS liquidations

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/ActivePool.sol#L169-L175

```
function sendColl(address _account, uint256 _amount) external override {
    _requireCallerIsBOorTroveMorSP();

    _accountForSendColl(_amount);
    collToken.safeTransfer(_account, _amount);
}
```

This could happen when some debt is being absorbed but 0 collateral is left in the position (e.g. when a liquidation pays out exclusively the coll incentive to the liquidator)

I have reviewed all Collaterals in scope and was unable to find any that would revert

Nonetheless you should keep this in mind as a key factor when evaluating collaterals

Code Snippet

Tool Used

Manual Review

Recommendation

None of the tokens in scope during the review will revert on 0 transfer If you end up changing the tokens, please do ensure that no tokens present this behaviour with fuzz testing

Alternatively you could fix this by applying a conditional check

The logic is used in `TroveManager` so this may be a small oversight

https://github.com/NeriteOrg/nerite/blob/b957c0417697ac4ee9c4aa6d4753739817f169df/contracts/src/TroveManager.sol#L439-L441

```
if (totals.collSurplus > 0) {
    activePoolCached.sendColl(address(collSurplusPool), totals.collSurplus);
}
```

Discussion

cupOJoseph

Acknowledged. This should not effect us for now.

Issue L-8: Trophy: Total Debt and Sum of Total Debts diverges over time

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/96

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

Summary

This is technically a known issue: https://github.com/liquity/bold?tab=readme-ov-file#7---discrepancy-between-aggregate-and-sum-of-individual-debts

I found it again while building my own invariant testing suite

Vulnerability Detail

The sum of debts and their weights, vs the sum of total weights diverges over time due to rounding errors

```
 // forge test --match-test test_property_CP01_1 -vvv
function test_property_CP01_1() public {

    borrowerOperations_openTrove_clamped(0x0000000000000000000000000000000000000000
    ,0,1512581343914037692,2002183030121286800210,0x0000000000000000000000000000000
    000000000,0x0000000000000000000000000000000000000000,0x000000000000000000000000
    0000000000000000);

    borrowerOperations_addColl_clamped(1031572922974545357689410);

    borrowerOperations_openTrove(0x00000000000000000000000000000000DeaDBeef,2353341
    6243,1103827468646747854,20000000000000000001,1412471596246372711115939,761236
    5641134122758440730,6219639325144576,2093495143961405296,0x00000000000000000000
    00000000000000000000,0x0000000000000000000000000000000000000000,0x0000000000000
    000000000000000000000000000);

    troveManager_liquidate_with_oracle_clamped();

    console2.log("_after.entireSystemDebt", _after.entireSystemDebt);
    console2.log("_after.ghostDebtAccumulator", _after.ghostDebtAccumulator);

    property_CP01();

 }

[PASS] test_property_CP01_1() (gas: 2752542)
Logs:
  _after.entireSystemDebt 4008062626152425153213
  _after.ghostDebtAccumulator 4008062626152424144619

/// @audit 1e6 discrepancy
```

I run a maximization test and the highest value I was able to generate is 200e6, while technically the debt diverges over time, the impact is negligible

https://getrecon.xyz/shares/3727e852-d15c-4cd0-80c2-22be0dc84deb

Impact

No particular impact is present given how small the discrepancy is

Tool Used

Manual Review

Recommendation

The finding requires no fixes

Discussion

cupOJoseph

Acknowledged.

Issue L-9: Pausable or Blacklist tokens may cause liquidations to fail

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/97

Summary

tETH and XVS present blacklisting logic that could break liquidations. XVS is also pausable, possibly leading to the same issue.

Vulnerability Detail

Liquidations in Nerite push collateral tokens to the `msg.sender`, from the Active Pool and to the StabilityPool.

If any of those 3 addresses is blacklisted or the collateral token is paused, liquidations that would normally work, can be DOSsed.

Impact

DoSed liquidations due to blacklisted addresses or paused tokens.

Code Snippet

XVS tETH

For example, for XVS, it is a standard OZ ERC20 with the following alteration to `_beforeTokenTransfer`:

```
/**
 * @notice Hook that is called before any transfer of tokens. This includes
 * minting and burning.
 * @param from_ Address of account from which tokens are to be transferred.
 * @param to_ Address of the account to which tokens are to be transferred.
 * @param amount_ The amount of tokens to be transferred.
 * @custom:error AccountBlacklisted is thrown when either `from` or `to` address is
↪  blacklisted.
 */
function _beforeTokenTransfer(address from_, address to_, uint256 amount_) internal
↪  override whenNotPaused {
    if (_blacklist[to_]) {
        revert AccountBlacklisted(to_);
    }
    if (_blacklist[from_]) {
        revert AccountBlacklisted(from_);
    }
}
```

Tool Used

Manual Review

Recommendation

These risks are not really possible to completely mitigate, unless other tokens are used.

Discussion

cupOJoseph

Acknowledged. Risk parameters for each token will be chosen based on this.

XVS has been removed as a col type

Issue L-10: MultiTroveGetter has a pending fix

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/98

Summary

MultiTroveGetter is a convenience contract

A PR with a fix is open in the Bold repo: https://github.com/liquity/bold/pull/817

Vulnerability Detail

The fix seems to send fresher data by using `getLatestTroveData` to grab the atest data

Impact

Informational

Code Snippet

Tool Used

Manual Review

Recommendation

Ask the Bold team and most likely merge the fix

Discussion

cupOJoseph

fixed using Liquity's code.

https://github.com/NeriteOrg/nerite/pull/57

cupOJoseph

reviewed. this is still the correct implementation
https://github.com/liquity/bold/blob/main/contracts/src/MultiTroveGetter.sol

Issue L-11: Swappers for Ramses and Camelot are missing

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/99

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

Summary

The scope of the review detailed a Ramses and a Camelot integration

These systems use a ”Velodrome” style interface

The code in scope doesn't seem to integrate with them

Recommendation

Write integrations and audit them extnesively

Discussion

cupOJoseph

Zaps will be updated for correct interfaces and are outside of the core protocol so they can be updated at any time. This is outside the scope of this audit for now.

Issue L-12: tBTC could have its MCR reduced to 115

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/103

Summary

tBTC's peg seems to be performing very well

Given strict borrow limits, it should be possible to allow more leverage than the provided 130 MCR would offer

Vulnerability Detail

Per the Recon analysis:
https://gist.github.com/Simon-Busch/9b4ac3669974499aa6afdb656408cf2a

tBTC has had it's worst price swing in one hour of just 5%

This is 3 times lower than ETH

Due to the imprecision in our analysis we're not suggesting lowering the CR down to 110, which can create tail risk

But given the fact that Nerite implements borrow limits

Given a sufficiently sound (based on liquidity available for liquidation) borrow cap, then tBTC could have it's MCR lowered to 115 as to offer more leverage to borrowers

Tool Used

Manual Review

Recommendation

Consider lowering the MCR for tBTC to 115 after discussing with Economic / Risk consultants

Discussion

cupOJoseph

done.

Issue L-13: Nerite borrow caps are directionally correct, however Arbitrum liquidity tends to change very quickly

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/104

Summary

Arbitrum liquidity changes very quickly, requiring a TWAP and statistical approach to caps that we are unable to provide

Governance should be made aware and should chose conservative caps as to respond to liquidity crunches

Vulnerability Detail

These are ballpark estimates taken on Friday 14th of Feb, 2025:

```
ETH 27 MLN before 5% price loss
WSTETH 1500 = 4.8 MLN before (15%) price loss
rETH 500 = 1.5 MLN = 26%
ARB 15MLN = $7.5 before 12% price change
```

As you can see in an analysis we did just a couple of weeks ago, these values are wildly different from the ones we got then: https://gist.github.com/Simon-Busch/9b4ac36699 74499aa6afdb656408cf2a#estimated-liquidity

Given the fact that only a % of total borrows should ever be liquidated (calculatable via a weighted TCR)

Then the borrow caps seem to be mostly correct

A methdology to asses borrow caps can be:

- Get the TWAP of the liquidity over the last X days / weeks
- Reduce by a risk factor based on how centralized and unlocked that liquidity is (one whale holding 100% of liquidity should be considered usnafe)
- Then multiply the cap by the average TCR as ultimately only a portion of the debt will need to ever be liquidated given a swing of Y% of prices

The following can be mostly made public and be voted on each week / biweekly by governance

Tool Used

Manual Review

Recommendation

Monitor Arbitrum liquidity as it seems to be very volatile

Discussion

cupOJoseph

Acknowledged, available liquidity will be monitored by the Nerite risk analysis team before raising caps.

Issue L-14: Redundant factory is passed in both constructor and initializer

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/105

Summary

The `constructor` and `initialize` code for `BoldToken` is as follows:

https://github.com/GalloDaSballo/nerite-fuzz/blob/8a3e4afc1060876a69c3ee8040e533d6a7a17cc3/contracts/src/BoldToken.sol#L57-L72

```
constructor(address _owner, ISuperTokenFactory factory) Ownable(_owner) {}

function initialize(ISuperTokenFactory factory) external {
    // This call to the factory invokes `UUPSProxy.initialize`, which connects the
 ↪  proxy to the canonical SuperToken implementation.
    // It also emits an event which facilitates discovery of this token.
    ISuperTokenFactory(factory).initializeCustomSuperToken(address(this));

    // This initializes the token storage and sets the `initialized` flag of
 ↪  OpenZeppelin Initializable.
    // This makes sure that it will revert if invoked more than once.
    ISuperToken(address(this)).initialize(
        IERC20(address(0)),
        18,
        _NAME,
        _SYMBOL
    );
}
```

They both receive the `ISuperTokenFactory factory`

The fact that the parameter used is the one in `intialize` opens up to #94

Changing the code to set the factory as an immutable parameter in the constructor, cleans up the code and fixes the issue

Code Snippet

Tool Used

Manual Review

Recommendation

Pass the factory in the constructor and then use that value in `initialize`

Discussion

cupOJoseph

Moved to only initialize function.

removed redundant param in constructor https://github.com/NeriteOrg/nerite/pull/55

Issue L-15: It’s best to provide some initial collateral on each branch to avoid long tail gotchas

Source: https://github.com/sherlock-audit/2025-02-nerite/issues/106

Summary

A few gotchas are present in the Bold system when no other trove is open

Performing a ”Deploy and Seed” strategy can prevent these underexplored edge cases

Vulnerability Detail

- Shutdown of Branch via one borrower triggering the Critical Threshold and never repaying
- Inability to liquidate said borrower in lack of another trove
- It may be possible for collateral stakes to be manipulated in lack of other depositors

Impact

Due to the time commitment and low likelihood of these, these are underexplored scenarios

Having one early depositor completely nullifies these scenarios At close to no cost

Code Snippet

Tool Used

Manual Review

Recommendation

Alter your deployment scripts to deploy, open a trove in each branch with a high CR

Discussion

cupOJoseph

Acknowledged. This is part of our deployment check list. Each branch will be seeded with a minimum of $200 of liquidity.

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.

This report has been signed off by 0x73696d616f.