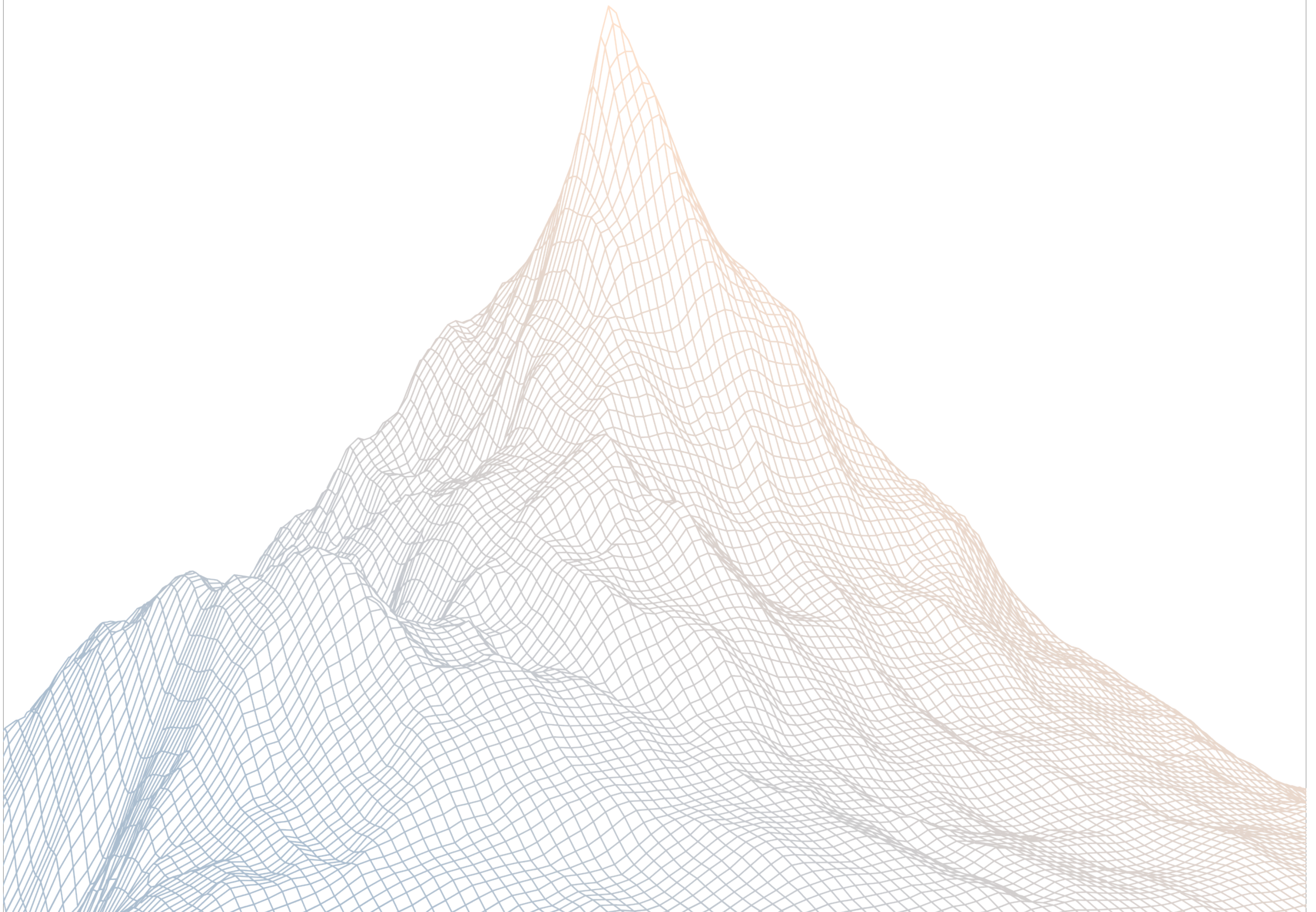
Zenith

Yala

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: February 13th to February 24th, 2025

AUDITED BY: 10xhash
said

Contents

1 Introduction 2

1.1 About Zenith 3

1.2 Disclaimer 3

1.3 Risk Classification 3

2 Executive Summary 3

2.1 About Yala 4

2.2 Scope 4

2.3 Audit Timeline 6

2.4 Issues Found 6

3 Findings Summary 6

4 Findings 8

4.1 Critical Risk 9

4.2 High Risk 16

4.3 Medium Risk 22

4.4 Low Risk 34

4.5 Informational 58

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Yala

Yala is a DeFi ecosystem designed to connect the Bitcoin liquidity to various sources of yields with a cross-chain strategy.

2.2 Scope

The engagement involved a review of the following targets:

Target	yala-core
Repository	https://github.com/yalaorg/yala-core/
Commit Hash	`db4ff1248d7017a53ad3304222ba54296a0c6fa0`

Files	`core/BorrowerOperations.sol`
	`core/DebtToken.sol`
	`core/Factory.sol`
	`core/GasPool.sol`
	`core/PSM.sol`
	`core/PriceFeed.sol`
	`core/RepayerDelegator.sol`
	`core/StabilityPool.sol`
	`core/TroveManager.sol`
	`core/YalaCore.sol`
	`dependencies/DelegatedOps.sol`
	`dependencies/EnumberableCollateral.sol`
	`dependencies/YalaBase.sol`
	`dependencies/YalaMath.sol`
	`dependencies/YalaOwnable.sol`
	`utils/MultiTroveGetters.sol`

Target	yala-yield-farming
Repository	https://github.com/yalaorg/yala-yield-farming
Commit Hash	`6f62be436641ad5d6518ac7314ad76aee1770ab3`

Files	`core/V3LPStaking.sol`
	`core/V3LPStakingHelpers.sol`
	`dependencies/EnumerableStake.sol`
	`dependencies/PoolADdress.sol`

Target	yala-notary
Repository	https://github.com/yalaorg/yala-notary
Commit Hash	`d7b1c5f5743e9c23dbd3d2132260d721debab33d`

Files	`BridgeController.sol`
	`BridgeToken.sol`
	`NotaryBridge.sol`
	`NotaryCustoryBridge.sol`

2.3 Audit Timeline

February 13, 2025	Audit start
February 24, 2025	Audit end
March 4, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	3
High Risk	3
Medium Risk	7
Low Risk	12
Informational	2
Total Issues	**27**

3

Findings Summary

ID	Description	Status
C-1	Re-entrancy in V3LPStaking allows an attacker to disallow withdrawals	Resolved
C-2	Calling ‘troveManager‘’s ‘accrueInterests‘ directly will not distribute ‘yieldSP‘ to ‘StabilityPool‘ users.	Resolved
C-3	Free debt token minting in PSM when the peg token uses non-18 decimals.	Resolved
H-1	‘BorrowerOperations‘’s ‘openTrove‘ not using latest trove state when calculating ‘TCR‘	Resolved
H-2	Lack of liquidation fees allows an attacker to self liquidate for profit	Resolved
H-3	uint96 casting for scaled price can lead to invalid prices	Resolved
M-1	Users can immediately reset newly created CDPs	Resolved
M-2	Equal division of collateral can cause some deserving tokens to be lost	Acknowledged
M-3	Collateral surplus is considered as part of totalActiveCollateral causing mis-representation of the TCR	Resolved
M-4	Gas compensation is unbacked	Resolved
M-5	Interest is not accrued before parameter updates	Resolved
M-6	Rounding down for ‘totalInterest‘ can cause withdrawals to revert	Resolved
M-7	Chainlink’s previous round fetching doesn’t consider phase changes	Resolved
L-1	Inconsistency between batch and sequential liquidation	Acknowledged

ID	Description	Status
L-2	Incorrect error factoring can cause a portion of withdrawals to be stuck	Resolved
L-3	ScaleUpdated event is emitted incorrectly	Resolved
L-4	Debt tokens are not burned when offesting	Resolved
L-5	Incorrect event emission when withdrawing	Resolved
L-6	BridgeController always assumes e18 decimals for delay classification	Resolved
L-7	Negative values for prices are currently allowed	Resolved
L-8	TCR restrictions on withdrawals can cause dependence on other positions	Acknowledged
L-9	Debt repayments cannot be made in case protocol is paused	Resolved
L-10	Short circuiting interest rewards is flawed	Resolved
L-11	‘maxSeizedColl‘ value is rounded down in favor of defaulters	Acknowledged
L-12	Defaulted debt and interest is not updated in case collateral value remains same	Resolved
I-1	computeNewStake always return 1:1 ratio	Acknowledged
I-2	User can bypass ‘buy‘ and ‘sell‘ fee in ‘PSM‘	Resolved

4 Findings

4.1 Critical Risk

A total of 3 critical risk findings were identified.

[C-1] Re-entrancy in V3LPStaking allows an attacker to disallow withdrawals

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- V3LPStaking.sol

Description:

The `V3LPStaking` contract invokes `safeTransferFrom` before clearing the tokens information from `stakes` mapping

```
    function unstake(uint256 id) external override {
        IUniswapV3Pool pool = _getPool(id);
        require(contains(msg.sender, pool, id), "not staked yet");
        EnumerableStake.Info memory info = stakes[msg.sender][pool].get(id);
        require(block.timestamp > info.unlockAt, "not unlocked yet");
⇒   positionManager.safeTransferFrom(address(this), msg.sender, id);
        stakes[msg.sender][pool].remove(id);
        totalStakes[info.pool] -= info.amount;
        emit Unstaked(msg.sender, id);
```

`safeTransferFrom` of ERC721 passes the execution on to the receiver if the receiver is a contract. This allows an attacker to first transfer back the assets to the `V3LPStaking` contract and then to re-enter and invoke unstake function on `V3LPStaking` with the same tokenId. This can be repeated how much ever times needed and will reduce the totalStakes of the pool. Following this, other legit stakers of the pool won't be able to unstake because it will underflow when performing the stake subtraction

Recommendations:

Perfrom the NFT transfer at the end

```
    function unstake(uint256 id) external override {
        IUniswapV3Pool pool = _getPool(id);
        require(contains(msg.sender, pool, id), "not staked yet");
        EnumerableStake.Info memory info = stakes[msg.sender][pool].get(id);
        require(block.timestamp > info.unlockAt, "not unlocked yet");
        stakes[msg.sender][pool].remove(id);
        totalStakes[info.pool] -= info.amount;
        emit Unstaked(msg.sender, id);
⇒       positionManager.safeTransferFrom(address(this), msg.sender, id);
```

Yala: Resolved at V3LPStaking.sol#L80

Zenith: Verified.

[C-2] Calling `troveManager's` `accrueInterests` directly will not distribute `yieldSP` to `StabilityPool` users.

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- TroveManager.sol#L494-L517
- StabilityPool.sol#L402

Description:

Inside `TroveManager`, it is possible to trigger `accrueInterests` directly. Additionally, `accrueInterests` is triggered during several operations that require the latest state of `TroveManager`.

```
    function accrueInterests() public returns (uint256 yieldSP,
    uint256 yieldFee) {
        (uint256 applicable, uint256 mintAmount) = getPendingInterest();
        if (lastInterestUpdate == 0) {
            lastInterestUpdate = applicable;
            return (0, 0);
        }
        if (mintAmount > 0) {
            uint256 interestNumerator = (mintAmount * DECIMAL_PRECISION)
    + lastActiveInterestError_Redistribution;
            uint256 interestRewardPerUnit = interestNumerator
    / totalActiveDebt;
            lastActiveInterestError_Redistribution = interestNumerator
    - totalActiveDebt * interestRewardPerUnit;
            L_active_interest += interestRewardPerUnit;
            yieldFee = (totalActiveDebt * interestRewardPerUnit)
    / DECIMAL_PRECISION;
            totalActiveInterest += yieldFee;
            if (SP_YIELD_PCT > 0) {
                yieldSP = (yieldFee * SP_YIELD_PCT) / DECIMAL_PRECISION;
                yieldFee -= yieldSP;
>>>     debtToken.mint(address(stabilityPool), yieldSP);
                emit SPYieldAccrued(yieldSP);
            }
```

```
                debtToken.mint(YALA_CORE.feeReceiver(), yieldFee);
                lastInterestUpdate = applicable;
                emit InterestAccrued(yieldFee);
            }
        }
```

Inside `accrueInterests`, if `SP_YIELD_PCT` is not 0, `yieldSP` is calculated and minted to the `stabilityPool`. However, if `accrueInterests` is called directly, the minted `yieldSP` will not be distributed to the `StabilityPool`'s depositors.

```
function _accrueAllYieldGains() internal {
    uint256 length = collateralTokens.length();
    uint256 _newYield = 0;
    for (uint256 i = 0; i < length; i++) {
        (address troveManager, ) = collateralTokens.at(i);
        (uint256 yieldSp, ) = ITroveManager(troveManager).accrueInterests();
        if (yieldSp > 0) {
            _newYield = _newYield + yieldSp;
        }
    }
    if (_newYield > 0) {
        uint256 accumulatedYieldGains = yieldGainsPending + _newYield;
        if (accumulatedYieldGains == 0) return;
        uint256 totalDebtTokenDepositsCached = totalDebtTokenDeposits; //
    cached to save an SLOAD
        if (totalDebtTokenDepositsCached < DECIMAL_PRECISION) {
            yieldGainsPending = accumulatedYieldGains;
            return;
        }
        yieldGainsPending = 0;
        uint256 yieldNumerator = accumulatedYieldGains * DECIMAL_PRECISION
    + lastYieldError;
        uint256 yieldPerUnitStaked = yieldNumerator
    / totalDebtTokenDepositsCached;
        lastYieldError = yieldNumerator - yieldPerUnitStaked
    * totalDebtTokenDepositsCached;
        uint256 marginalYieldGain = yieldPerUnitStaked * (P - 1);
        epochToScaleToG[currentEpoch][currentScale]
    = epochToScaleToG[currentEpoch][currentScale] + marginalYieldGain;
        emit G_Updated(epochToScaleToG[currentEpoch][currentScale],
    currentEpoch, currentScale);
    }
}
```

It can be seen that inside `StabilityPool`, `yieldSp` is distributed under the assumption that `TroveManager`'s `accrueInterests` is only called when `_accrueAllYieldGains` is triggered. If

`accrueInterests` is called outside of `_accrueAllYieldGains`, the yield will not be distributed.

Recommendations:

Inside `StabilityPool`, consider adding a function to distribute the yield and allowing it to be called by `TroveManager` right after `yieldSP` is minted within `TroveManager`'s `accrueInterests`.

Yala: Resolved with @a7729496c6...

Zenith: Verified.

[C-3] Free debt token minting in PSM when the peg token uses non-18 decimals.

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- PSM.sol#L85-L88

Description:

When users call `buy` inside `PSM` and provide `amountDebtToken` they want to mint, it calls `estimateBuy` to calculate the `amountPegTokenUsed` that users have to pay.

```
function buy(uint256 amountDebtToken)
    external override whenNotPaused returns (uint256 amountPegTokenUsed,
    uint256 fee) {
    require(amountDebtToken > 0, "PSM: Amount debt token must be greater than
    0");
    require(totalActivedebt + amountDebtToken <= supplyCap, "PSM: Supply cap
    reached");
    totalActivedebt = totalActivedebt + amountDebtToken;
>>> (amountPegTokenUsed, fee) = estimateBuy(amountDebtToken);
    IERC20(pegToken).safeTransferFrom(msg.sender, address(this),
    amountPegTokenUsed);
    debtToken.mint(msg.sender, amountDebtToken);
    if (fee > 0) debtToken.mint(YALA_CORE.feeReceiver(), fee);

    emit Buy(msg.sender, amountDebtToken, amountPegTokenUsed, fee);
}
```

Inside `estimateBuy`, it can be seen that when calculating `amountPegTokenUsed`, it uses `priceFactor` to scale the amount that needs to be paid, depending on the peg token's decimals.

```
    function estimateBuy(uint256 amountDebtToken)
    public view override returns (uint256 amountPegTokenUsed, uint256 fee) {
        fee = (amountDebtToken * feeIn) / DECIMAL_PRECISIONS;
>>>  amountPegTokenUsed = (amountDebtToken + fee) / priceFactor;
```

```
    }
```

For instance, if the peg token's decimals are 8, the `priceFactor` will be `10 ** (18 - pegTokenDecimals) = 10 ** 10`. This means that if a user provides an `amountDebtToken` where `amountDebtToken + fee < priceFactor`, causing `amountPegTokenUsed` to become 0, they can mint the `amountDebtToken` for free.

Recommendations:

Consider redesigning the `buy` function, allow users to provide the amount of peg tokens they want to convert to debt tokens and calculate `amountDebtToken` accordingly to avoid rounding issues.

Yala: Resolved with the following [commit]

Zenith: Verified.

4.2 High Risk

A total of 3 high risk findings were identified.

[H-1] `BorrowerOperations`'s `openTrove` not using latest trove state when calculating `TCR`

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- BorrowerOperations.sol#L76
- TroveManager.sol#L256-L270

Description:

When `openTrove` is called, it will call `getCollateralAndTCRData` to get provided `troveManager`'s `totalCollateral`, `totalDebt` and `totalInterest`, which will be used to calculate `TCR` and `newTCR`.

```
    function openTrove(ITroveManager troveManager, address account,
    uint256 _collateralAmount, uint256 _debtAmount)
    external callerOrDelegated(account) returns (uint256 id) {
        require(!YALA_CORE.paused(), "BorrowerOps: Deposits are paused");
        LocalVariables_openTrove memory vars;
        vars.netDebt = _debtAmount;
        vars.compositeDebt = _getCompositeDebt(vars.netDebt);
        _requireAtLeastMinNetDebt(vars.compositeDebt);
>>>  (vars.collateralToken, vars.totalCollateral, vars.totalDebt,
    vars.totalInterest, vars.price)
    = _getCollateralAndTCRData(troveManager);
        (vars.MCR, vars.CCR) = (troveManager.MCR(), troveManager.CCR());
        vars.ICR = YalaMath._computeCR(_collateralAmount,
    vars.compositeDebt, vars.price);
        _requireICRisAboveMCR(vars.ICR, vars.MCR);
>>>  uint256 TCR = YalaMath._computeCR(vars.totalCollateral, vars.totalDebt
    + vars.totalInterest, vars.price);
        if (TCR >= vars.CCR) {
```

```
>>>   uint256 newTCR = _getNewTCRFromTroveChange(vars.totalCollateral
    * vars.price, vars.totalDebt + vars.totalInterest, _collateralAmount
    * vars.price, true, vars.compositeDebt, true);
            _requireNewTCRisAboveCCR(newTCR, vars.CCR);
        } else {
            _requireICRisAboveCCR(vars.ICR, vars.CCR);
        }
        // Create the trove
        id = troveManager.openTrove(account, _collateralAmount,
    vars.compositeDebt);
        // Move the collateral to the Trove Manager
        vars.collateralToken.safeTransferFrom(msg.sender,
    address(troveManager), _collateralAmount);
        //  and mint the DebtAmount to the caller and gas compensation for
    Gas Pool
        debtToken.mintWithGasCompensation(account, vars.netDebt);
        emit TroveCreated(account, troveManager, id, _collateralAmount,
    vars.compositeDebt);
    }
```

However, it doesn't trigger `TroveManager.accrueInterests`, causing the returned `totalInterest` to be outdated and resulting in incorrect calculations of `TCR` and `newTCR`.

Recommendations:

Trigger `TroveManager.accrueInterests` before retrieving all the trove state information.

Yala: Resolved with @720460c164f...

Zenith: Verified.

[H-2] Lack of liquidation fees allows an attacker to self liquidate for profit

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TroveManager.sol

Description:

Currently there is no liquidation fee attached to liquidations. Hence on a liquidation, the entire amount of collateral is given back to the liquidator (JIT case) if they can provide `totalTakenDebt - gasCompensation amount` of debt tokens. This is flawed as this allows an attacker to drain yala by self liquidating

```
function batchLiquidate(uint256[] memory ids, bool isJIT)
    external whenNotPaused whenNotShutdown {

        ...

    debtToken.returnFromPool(gasPoolAddress, msg.sender,
    totals.debtGasCompensation);
    _sendCollateral(msg.sender, totals.collGasCompensation);
   }

function _JIT(address account, uint256 _coll, uint256 _debt,
    uint256 _interest) internal {
    if (_debt == 0 && _interest == 0) {
        return;
    }
    totalActiveDebt -= _debt;
    totalActiveInterest -= _interest;
    debtToken.burn(account, _debt + _interest);
    _sendCollateral(account, _coll);
```

Eg: Currently stability pool has 0 tokens (ie. this is to make the JIT liquidator liquidate the entire position) Attacker opens a position just below liquidation threshold with 100 collateral, 80 debt token. Out of this, 5 debt tokens will be sent to the gas pool In the next

block it becomes liquidateable (either via interest update or via price change. attacker can be guarded against other liquidators by prebooking transaction slot in someway) Attacker liquidates his own position. Now 100 collateral will be returned to him and 75 debt tokens will be burned from him. The rest 5 will be sent to him as debt gas compensation. These are free tokens. Attacker can open up several such positions to mint large amount of yala tokens

Recommendations:

Enforcing a liquidation fee should prevent the above attack. But needs to think further about the best way

Yala: Resolved with @1bdd4c4e7d...

Zenith: Verified.

[H-3] uint96 casting for scaled price can lead to invalid prices

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- PriceFeed.sol
- PriceFeed.sol

Description:

Prices are downcasted to uint96 while being scaled in order to maintain `token decimals + price decimals == 36`

```
    function _storePrice(address _token, uint256 _price, uint256 _timestamp,
    uint80 roundId) internal {
⇒   priceRecords[_token] = PriceRecord({ scaledPrice: uint96(_price),
    timestamp: uint32(_timestamp), lastUpdated: uint32(block.timestamp),
    roundId: roundId });
    }
```

```
    function _setOracle(address _token, address _chainlinkOracle,
    uint32 _heartbeat) internal {
        uint8 decimals = IERC20Metadata(_token).decimals();
        if (decimals > 18) {
            revert PriceFeed__UnsupportedTokenDecimalsError(_token,
    decimals);
        }
⇒   TARGET_DIGITS[_token] = MAX_DIGITS - decimals;
```

This is flawed as the scaled price can overflow uint96 for lower decimal tokens

Eg:

```
function testUint96() public {
        // asset == WBTC
        uint price=9782192244323;
        uint assetDecimlas = 8;
```

```
        uint maxDecimals = 36;
        uint oracleDecimals = 8;
        uint target = maxDecimals - assetDecimlas;

        uint scaledPrice = (price) * (10 ** (target - oracleDecimals));
        assert(scaledPrice > type(uint96).max);
    }
```

Recommendations:

Make prices uint256 itself

Yala: Resolved with @527bfd6acfe...

Zenith: Verified.

4.3 Medium Risk

A total of 7 medium risk findings were identified.

[M-1] Users can immediately reset newly created CDPs

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TroveManager.sol#L455-L464

Description:

When a new CDP is created, `TroveManager` will be instantiated and configured using the parameters provided by the owner when calling `deployNewCDP`.

```
function setParameters(IPriceFeed _priceFeed, IERC20 _collateral,
    DeploymentParams memory params) external {
    require(address(collateralToken) == address(0) && msg.sender ==
    factoryAddress, "TM: parameters set");
    require(params.interestRate < DECIMAL_PRECISION, "TM: interest rate too
    high");
    require(params.maxDebt > 0, "TM: interest rate too high");
    require(params.spYieldPCT <= DECIMAL_PRECISION, "TM: sp yield pct too
    high");
    require(params.MCR > DECIMAL_PRECISION && params.SCR >= params.MCR &&
    params.CCR >= params.SCR, "TM: invalid cr parameters");
    collateralToken = _collateral;
    _setPriceFeed(_priceFeed);
    _setMetadataNFT(params.metadataNFT);
    _setInterestRate(params.interestRate);
    _setMaxSystemDebt(params.maxDebt);
    _setSPYielPCT(params.spYieldPCT);
    _setMaxCollGasCompensation(params.maxCollGasCompensation);
    _setLiquidationPenaltySP(params.liquidationPenaltySP);
    _setLiquidationPenaltyRedist(params.liquidationPenaltyRedistribution);
    MCR = params.MCR;
```

```
    SCR = params.SCR;
    CCR = params.CCR;
    emit CRUpdated(MCR, SCR, CCR);
}
```

When the last user of the `TroveManager` closes the trove, it will `_resetState`, which includes setting `maxSystemDebt`, `interestRate`, and `SP_YIELD_PCT` to 0, effectively preventing anyone from opening a trove using this `TroveManager`.

```
function _resetState() private {
    if (totalSupply() == 0) {
        maxSystemDebt = 0;
        interestRate = 0;
        SP_YIELD_PCT = 0;
        totalStakes = 0;
        totalStakesSnapshot = 0;
        totalCollateralSnapshot = 0;
        L_collateral = 0;
        L_debt = 0;
        L_defaulted_interest = 0;
        L_active_interest = 0;
        lastCollateralError_Redistribution = 0;
        lastDebtError_Redistribution = 0;
        lastActiveInterestError_Redistribution = 0;
        lastDefaultedInterestError_Redistribution = 0;
        totalActiveCollateral = 0;
        totalActiveDebt = 0;
        totalActiveInterest = 0;
        defaultedCollateral = 0;
        defaultedDebt = 0;
        defaultedInterest = 0;
        lastInterestUpdate = 0;
        nonce = 0;
    }
}
```

This opens a grief vector: when a new `TroveManager` is created, an attacker can create and immediately close the trove to trigger `_resetState`, preventing users from creating new troves until the `TroveManager` is reconfigured.

Recommendations:

When `_resetState` is called, consider to not reset `maxSystemDebt`, `interestRate`, `SP_YIELD_PCT`.

Yala: Resolved with @1fb06a6f2a0...

Zenith: Verified.

[M-2] Equal division of collateral can cause some deserving tokens to be lost

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- TroveManager.sol

Description:

Currently the collateral division b/w the stability pool and the second method doesn't factor in the different possible liquidation penalties. They are divided equally based on the debt amount they are taking in

```
    function _liquidatePenalty(LiquidationValues memory totals,
    SingleLiquidation memory singleLiquidation, uint256 _price, bool isJIT)
    internal view {

        ...

=>   collSPPortion = (singleLiquidation.collToLiquidate
    * (singleLiquidation.debtOffset + singleLiquidation.interestOffset))
    / (singleLiquidation.debtToLiquidate + singleLiquidation.interest);

        }
```

This causes scenario's where the amount supposed to be received by stability pool can go the defaulter as collateral surplus

Eg: debt = 100 collateral = 150 liquidation penalty SPool = 1.6 liquidation penalty JIT = 1.4

assume spool has debt == 50. Now collateral will be split as 75,75 This is loss of 5 for the spool (50 * 1.6 == 80) and an excess of 5 for the JIT (50 * 1.4 == 70). Hence this amount will go to the defeulter instead of the spool while the amount was enough to settle both parties ideally

Recommendations:

Need to consider a method which factors in the different liquidation penalties as well

Yala: Acknowledged

[M-3] Collateral surplus is considered as part of totalActiveCollateral causing mis-representation of the TCR

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TroveManager.sol

Description:

Collateral surplus from liquidations is included in the `totalActiveCollateral` which is used for TCR calculations. This is problematic because these assests can be freely moved anytime which will decrase the TCR. Hence the values given by TCR won't be reliable to asses the risk of the protocol before determining the ratio in which trove adjustments can be made

```
function claimCollSurplus(address account, uint256 _amount) external {
    require(accountCollSurplus[account] >= _amount, "TM: insufficient coll
    surplus");
    accountCollSurplus[account] -= _amount;
    _sendCollateral(account, _amount);
    emit CollSurplusClaimed(account, _amount);
}
```

Recommendations:

Don't include collateral surplus in totalActiveInterest

Yala: Resolved with @9d692df85e...

Zenith: Verified.

[M-4] Gas compensation is unbacked

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TroveManager.sol

Description:

Liquidators are given out gas compensations. But this amount is not backed/accounted by anything (eg: like a system-wide bad debt). Hence these amount of tokens will cause the overall value of the stable yala token to be lower

```
function batchLiquidate(uint256[] memory ids, bool isJIT)
    external whenNotPaused whenNotShutdown {

       ....

    require(totals.debtGasCompensation > 0, "TM: nothing to liquidate");
    totalActiveInterest = totalActiveInterest - totals.interestOffset;
    totalActiveDebt = totalActiveDebt - totals.debtGasCompensation
    - totals.debtOffset;

       ....

    debtToken.returnFromPool(gasPoolAddress, msg.sender,
    totals.debtGasCompensation);
```

Recommendations :

Develop a mechanism to deal with this amount eg:maintain a system-wide bad debt and adjust it with the yield

Yala: Resolved with @ed79a2a9e4...

Zenith: Verified.

[M-5] Interest is not accrued before parameter updates

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TroveManager.sol

Description:

Multiple functions like `setInterestRate`, `setSPYielPCT` function doesn't invoke the accrueInterests function before updating to their new value. Hence the new interest rate will be applied for the entire period beggining from the last update rather than from the moment the update was made

```
function setInterestRate(uint256 _interestRate) external onlyOwner {
    _setInterestRate(_interestRate);
}

function _setInterestRate(uint256 _interestRate) internal {
    interestRate = _interestRate;
    emit InterestRateUpdated(_interestRate);
}

   function setSPYielPCT(uint256 _spYielPCT) external onlyOwner {
    _setSPYielPCT(_spYielPCT);
}
```

Recommendations:

Invoke the `accrueInterests` function before updating the parameters

Yala: Resolved at TroveManager.sol#L204

Zenith: Verified.

[M-6] Rounding down for `totalInterest` can cause withdrawals to revert

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- TroveManager.sol
- TroveManager.sol

Description:

When adding to the `totalInterest`, the current interest delta is rounded down. While for the individual trove positions, this need not be the case. This makes it possible for the totalInterest to be less than the sum of individual trove interests. One scenario where this can have a significant impact is when all the troves attempt to close (for eg: shutdown). The last trove will be unable to close because of underflow when attempting to subtact the trove's interest from the totalInterest

totalInterest rounds down https://github.com/yalaorg/yala-core/blob/db4ff1248d7017a53ad3304222ba54296a0c6fa0/contracts/core/TroveManager.sol#L5
L506

```
    function accrueInterests() public returns (uint256 yieldSP,
    uint256 yieldFee) {
        (uint256 applicable, uint256 mintAmount) = getPendingInterest();
        if (lastInterestUpdate == 0) {
            lastInterestUpdate = applicable;
            return (0, 0);
        }
        if (mintAmount > 0) {
            uint256 interestNumerator = (mintAmount * DECIMAL_PRECISION)
    + lastActiveInterestError_Redistribution;
            uint256 interestRewardPerUnit = interestNumerator
    / totalActiveDebt;
            lastActiveInterestError_Redistribution = interestNumerator
    - totalActiveDebt * interestRewardPerUnit;
            L_active_interest += interestRewardPerUnit;
=>   yieldFee = (totalActiveDebt * interestRewardPerUnit)
    / DECIMAL_PRECISION;
```

```
            totalActiveInterest += yieldFee;
```

individual trove's interest calculation need not always result in precision loss even if the totalInterest calculation had rounded down https://github.com/yalaorg/yala-core/blob/db4ff1248d7017a53ad3304222ba54296a0c6fa0/contracts/core/TroveManager.sol#L5 L527

```
function _accrueTroveInterest(uint256 id) internal returns (uint256 total,
    uint256 accrued) {
    accrueInterests();
    Trove storage t = Troves[id];
    if (rewardSnapshots[id].activeInterest < L_active_interest) {
        accrued = ((L_active_interest - rewardSnapshots[id].activeInterest)
    * t.debt) / DECIMAL_PRECISION;
        t.interest += accrued;
    }
    total = t.interest;
}
```

POC Test:

Add the following to `test/BorrowerOperations.test.ts`. It can be seen that closeTrove will revert with underflow

```
it.only('cannot close trove due to underflow', async () => {
    const troveManager = await deployNewCDP(fixture, { spYieldPCT:
    0n,interestRate: parseEther("0.07")})
    const { id } = await openTrove(fixture, { debt: 1800112548948470006153n,
    troveManager })
    const { BorrowerOperations, accounts, signers, DebtToken,
    MockCollateralToken } = fixture
    await time.increase(356)
    await troveManager.accrueInterests();

    await time.increase(356)
    await troveManager.shutdown()
    await time.increase(7000)

    await BorrowerOperations.closeTrove(troveManager, id, accounts[1])
})
```

Recommendations:

When closing the trove, limit subtraction to 0 to prevent the underflow

Yala: Resolved with @8b666bef2d... & @7255a2d052b...

Zenith: Verified.

[M-7] Chainlink's previous round fetching doesn't consider phase changes

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- PriceFeed.sol

Description:

The previousRoundId to fetch is always computed as `_currentRoundId - 1`. This is flawed as when a phase changes, the newRoundId will be (phaseId * 2^64 + 1) and the actual previous roundId would be ((phaseId - 1) * 2^64 + latestRoundIdOfPreviousPhase)

```
    function _fetchPrevFeedResponse(IAggregatorV3Interface _priceAggregator,
    uint80 _currentRoundId)
    internal view returns (FeedResponse memory prevResponse) {

        ...

⇒    try _priceAggregator.getRoundData(_currentRoundId - 1)
    returns (uint80 roundId, int256 answer, uint256 /* startedAt */,
    uint256 timestamp, uint80 /* answeredInRound */) {
            prevResponse.roundId = roundId;
```

Fetching round 0 will return 0 values which will cause the response to be rejected. This condition remains till the roundId increases. During this time period all the functionalities requiring price will be DOS'd

Recommendations:

Compare with the actual previous round id rather than always taking latest - 1

Yala: Resolved with @4dcc36bee1c...

Zenith: Verified.

4.4 Low Risk

A total of 12 low risk findings were identified.

[L-1] Inconsistency between batch and sequential liquidation

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- TroveManager.sol#L347-L372

Description:

There is a known issue in this codebase forks inside `batchLiquidate`, caused by bad debt redistribution happens at the end of operation, after every troves provided is processed.

```
function batchLiquidate(uint256[] memory ids, bool isJIT)
external whenNotPaused whenNotShutdown {
    uint256 price = fetchPrice();
    LiquidationValues memory totals;
    totals.remainingDeposits = stabilityPool.getTotalDeposits();
    for (uint256 i = 0; i < ids.length; i++) {
        uint256 id = ids[i];
        SingleLiquidation memory singleLiquidation;
        (singleLiquidation.coll, singleLiquidation.debt,
singleLiquidation.interest) = applyPendingRewards(id);
        _liquidate(id, totals, singleLiquidation, price, isJIT);
    }
    require(totalSupply() > 0, "TM: at least one trove to redistribute
coll and debt");
    require(totals.debtGasCompensation > 0, "TM: nothing to liquidate");
    totalActiveInterest = totalActiveInterest - totals.interestOffset;
    totalActiveDebt = totalActiveDebt - totals.debtGasCompensation
- totals.debtOffset;
    if (totals.debtOffset > 0) {
        _sendCollateral(address(stabilityPool), totals.collOffset);
        stabilityPool.offset(totals.debtOffset + totals.interestOffset,
totals.collOffset);
```

```
        }
        debtToken.returnFromPool(gasPoolAddress, msg.sender,
    totals.debtGasCompensation);
        _sendCollateral(msg.sender, totals.collGasCompensation);
        if (isJIT) {
            _JIT(msg.sender, totals.collRedistOrJIT, totals.debtRedistOrJIT,
    totals.interestRedistOrJIT);
        } else {
>>>     _redistribute(totals.collRedistOrJIT, totals.debtRedistOrJIT,
    totals.interestRedistOrJIT);
        }
    }
```

When a trove creates bad debt, the bad debt should be updated immediately, as it impacts the next trove's debt and collateral offset calculation. Otherwise, the next trove will process incorrect debt and collateral calculations, which will affect the system's health.

Recommendations:

Either Redistribute bad debt after each liquidation or accept the risk.

Yala: Acknowledged

[L-2] Incorrect error factoring can cause a portion of withdrawals to be stuck

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- StabilityPool.sol

Description:

The new P by scale function adds the previous loss in P percentage wise to the new P. But this is flawed as a deposit in b/w can cause the sum of positions debt's to be greater than the acually added debt tokens

```
function _getNewPByScale(uint256 _currentP, uint256 _newProductFactor,
    uint256 _lastDebtLossErrorByP_Offset,
    uint256 _lastDebtLossError_TotalDeposits, uint256 _scale)
    internal pure returns (uint256) {
    uint256 errorFactor;
    if (_lastDebtLossErrorByP_Offset > 0) {
        errorFactor = (_lastDebtLossErrorByP_Offset * _newProductFactor
    * _scale) / _lastDebtLossError_TotalDeposits / DECIMAL_PRECISION;
    }
    return (_currentP * _newProductFactor * _scale + errorFactor)
    / DECIMAL_PRECISION;
```

This can cause the final withdrawals to revert due to overflow

POC Test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";

contract IncorrectErrorOffsetAdditionTest is Test {
    uint totalDebtTokenDeposits;
    uint debtLoss1;
    uint debtLoss2;
```

```
uint addingAmountInBw;

uint P = 1e18;
uint pDeposit1;
uint pDeposit2;

uint constant DECIMAL_PRECISION = 1e18;
uint SCALE_FACTOR = 1e9;
uint128 currentScale;
uint128 currentEpoch;

struct Snapshots {
    uint256 P;
    uint128 scale;
    uint128 epoch;
}

uint lastDebtLossErrorByP_Offset;
uint lastDebtLossError_TotalDeposits;

mapping(address => Snapshots) userDeposits;

function inner_IncorrectAdditionOfErrorOffset(uint lossAmount1,uint
lossAmount2,uint userDeposit1,uint userDeposit2 ) internal {

    totalDebtTokenDeposits= userDeposit1;
    pDeposit1 = P;

    userDeposits[address(1)]
= Snapshots({P:P,scale:currentScale,epoch:currentEpoch});

   offset(lossAmount1);

console.log("lastDebtLossErrorByP_Offset",lastDebtLossErrorByP_Offset);

   totalDebtTokenDeposits += userDeposit2;
    pDeposit2 = P;

    userDeposits[address(2)]
= Snapshots({P:P,scale:currentScale,epoch:currentEpoch});
    offset(lossAmount2);

        uint user1WithdrawableAmount
= _getCompoundedDebtDeposit(address(1),userDeposit1);
        uint user2WithdrawableAmount
= _getCompoundedDebtDeposit(address(2),userDeposit2);
```

```
        console.log("userDeposit1",userDeposit1);
        console.log("lossAmount1",lossAmount1);
        console.log("pDeposit1",pDeposit1);
        console.log("userDeposit2",userDeposit2);
        console.log("lossAmount2",lossAmount2);
        console.log("pDeposit2",pDeposit2);
        console.log("totalDebtTokenDeposits",totalDebtTokenDeposits);
        console.log("user1WithdrawableAmount",user1WithdrawableAmount);
        console.log("user2WithdrawableAmount",user2WithdrawableAmount);
        console.log("diff",int(totalDebtTokenDeposits)
    - int(user1WithdrawableAmount+user2WithdrawableAmount));
        // -3515916998693767
        // -351590979023888
        assert(totalDebtTokenDeposits >= user1WithdrawableAmount
    + user2WithdrawableAmount);
    }

    function testSpecific_IncorrectAdditionOfErrorOffset () public {
       uint userDeposit1 = 5000000000000000003;
  uint lossAmount1 =4999999999999999997;
  uint userDeposit2 =1757958920326119919472530;
  uint lossAmount2 =4000000001677213174;

        inner_IncorrectAdditionOfErrorOffset(lossAmount1, lossAmount2,
    userDeposit1, userDeposit2);
    }

    function _getCompoundedDebtDeposit(address _depositor,
    uint256 initialDeposit) internal view returns (uint256) {
        if (initialDeposit == 0) {
            return 0;
        }
        Snapshots memory snapshots = userDeposits[_depositor];
        uint256 snapshot_P = snapshots.P;
        uint128 scaleSnapshot = snapshots.scale;
        uint128 epochSnapshot = snapshots.epoch;
        // If stake was made before a pool-emptying event, then it has been
    fully cancelled with debt -- so, return 0
        if (epochSnapshot < currentEpoch) {
            return 0;
        }
        uint256 compoundedStake;
        uint128 scaleDiff = currentScale - scaleSnapshot;
        uint256 cachedP = P;
        uint256 currentPToUse = cachedP != snapshot_P ? cachedP - 1 :
    cachedP;
        console.log("-1 performed",cachedP != snapshot_P);
```

```
        /* Compute the compounded stake. If a scale change in P was made
during the stake's lifetime,
         * account for it. If more than one scale change was made, then the
stake has decreased by a factor of
         * at least 1e-9 -- so return 0.
         */
        if (scaleDiff == 0) {
            compoundedStake = (initialDeposit * currentPToUse) / snapshot_P;
        } else if (scaleDiff == 1) {
            compoundedStake = (initialDeposit * currentPToUse) / snapshot_P
/ SCALE_FACTOR;
        } else {
            compoundedStake = 0;
        }

        if (compoundedStake < initialDeposit / 1e9) {
            return 0;
        }

        return compoundedStake;
    }


    function offset( uint256 _debtToOffset) public {
        uint256 totalDebt = totalDebtTokenDeposits; // cached to save an
SLOAD
        if (totalDebt == 0 || _debtToOffset == 0) {
            return;
        }
        _updateCollRewardSumAndProduct( _debtToOffset, totalDebt);
        _decreaseDebt(_debtToOffset);
        }

    function _decreaseDebt(uint256 _amount) internal {
        uint256 newTotalDebtTokenDeposits = totalDebtTokenDeposits
- _amount;
        totalDebtTokenDeposits = newTotalDebtTokenDeposits;
    }

    function _getNewPByScale(uint256 _currentP, uint256 _newProductFactor,
    uint256 _lastDebtLossErrorByP_Offset,
    uint256 _lastDebtLossError_TotalDeposits, uint256 _scale)
    internal pure returns (uint256) {
        uint256 errorFactor;
        if (_lastDebtLossErrorByP_Offset > 0) {
            // @bug this should be incorrect and should cause more than
    withdrawable tokens than there is the debt amount left (users depositing
```

```
will result in greater than _lastDebtLossError_TotalDeposits amount of
deposits)
        errorFactor = (_lastDebtLossErrorByP_Offset * _newProductFactor
* _scale) / _lastDebtLossError_TotalDeposits / DECIMAL_PRECISION;
        console.log("error factor",errorFactor);
        console.log("_newProductFactor",_newProductFactor);

    }
    console.log("_currentP",_currentP);
    return (_currentP * _newProductFactor * _scale + errorFactor)
/ DECIMAL_PRECISION;
}

function _updateCollRewardSumAndProduct(  uint256 _debtToOffset,
uint256 _totalDeposits) internal {
    ( uint256 debtLossPerUnitStaked, uint256 newLastDebtLossErrorOffset)
= _computeCollRewardsPerUnitStaked(_debtToOffset, _totalDeposits);

    uint256 currentP = P;
    uint256 newP;

    assert(debtLossPerUnitStaked <= DECIMAL_PRECISION);
    /*
     * The newProductFactor is the factor by which to change all
deposits, due to the depletion of Stability Pool Debt in the liquidation.
     * We make the product factor 0 if there was a pool-emptying.
Otherwise, it is (1 - debtLossPerUnitStaked)
     */
    uint256 newProductFactor = uint256(DECIMAL_PRECISION)
- debtLossPerUnitStaked;

    uint128 currentScaleCached = currentScale;
    uint128 currentEpochCached = currentEpoch;


    // If the Stability Pool was emptied, increment the epoch, and reset
the scale and product P
    if (newProductFactor == 0) {
        currentEpoch = currentEpochCached + 1;
        currentScale = 0;
        newP = DECIMAL_PRECISION;
    } else {
        uint256 lastDebtLossErrorByP_Offset_Cached
= lastDebtLossErrorByP_Offset;
        uint256 lastDebtLossError_TotalDeposits_Cached
= lastDebtLossError_TotalDeposits;
        newP = _getNewPByScale(currentP, newProductFactor,
```

```
lastDebtLossErrorByP_Offset_Cached,
lastDebtLossError_TotalDeposits_Cached, 1);
            console.log("newP ",newP);
            console.log("newProductFactor,",newProductFactor);

        // If multiplying P by a non-zero product factor would reduce P
below the scale boundary, increment the scale
        if (newP < SCALE_FACTOR) {
            console.log("newP lower than scale factor,",newP);
            newP = _getNewPByScale(currentP, newProductFactor,
lastDebtLossErrorByP_Offset_Cached,
lastDebtLossError_TotalDeposits_Cached, SCALE_FACTOR);
            currentScale = currentScaleCached + 1;
            console.log("newP afterwrds,",newP);

            // Increment the scale again if it's still below the
boundary. This ensures the invariant P ≥ 1e9 holds and
            // addresses this issue from Liquity v1:
https://github.com/liquity/dev/security/advisories/GHSA-m9f3-hrx8-x2g3
            if (newP < SCALE_FACTOR) {
            console.log("newP 2x lower than scale factor,",newP);

                newP = _getNewPByScale(currentP, newProductFactor,
lastDebtLossErrorByP_Offset_Cached,
lastDebtLossError_TotalDeposits_Cached, SCALE_FACTOR * SCALE_FACTOR);
                currentScale = currentScaleCached + 2;
            }
        }

    }
    lastDebtLossErrorByP_Offset = currentP * newLastDebtLossErrorOffset;
    lastDebtLossError_TotalDeposits = _totalDeposits;

    assert(newP > 0);
    P = newP;

}

function _computeCollRewardsPerUnitStaked( uint256 _debtToOffset,
uint256 _totalDebtDeposits) internal returns (
uint256 debtLossPerUnitStaked, uint256 newLastDebtLossErrorOffset) {


    assert(_debtToOffset <= _totalDebtDeposits);
    if (_debtToOffset == _totalDebtDeposits) {
        debtLossPerUnitStaked = DECIMAL_PRECISION; // When the Pool
depletes to 0, so does each deposit
```

```
            newLastDebtLossErrorOffset = 0;
        } else {
            uint256 debtLossNumerator = _debtToOffset * DECIMAL_PRECISION;
            /*
             * Add 1 to make error in quotient positive. We want "slightly
    too much" Debt loss,
             * which ensures the error in any given compoundedDebtDeposit
    favors the Stability Pool.
             */
            debtLossPerUnitStaked = debtLossNumerator / _totalDebtDeposits
    + 1;
            newLastDebtLossErrorOffset = debtLossPerUnitStaked
    * _totalDebtDeposits - debtLossNumerator;
        }


        return ( debtLossPerUnitStaked, newLastDebtLossErrorOffset);
    }
}
```

Recommendations:

Yala: Resolved with @495eab3174... & @1cba6dc54756...

Zenith: Verified

[L-3] ScaleUpdated event is emitted incorrectly

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- StabilityPool.sol

Description:

The `ScaleUpdated` event is emitted even when a scale update doesn't actually occur

```
emit ScaleUpdated(currentScale);
```

Recommendations:

The event should be emitted inside the if clause.

Yala: Resolved with @d47893427db...

Zenith: Verified.

[L-4] Debt tokens are not burned when offesting

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- StabilityPool.sol

Description:

When offsetting the debt from stability pool, the amount is only adjusted to not be claimable by the users but is not burned. This causes the totalSupply of the stable yala token to be higher than the backing value

```
function _decreaseDebt(uint256 _amount) internal {
    uint256 newTotalDebtTokenDeposits = totalDebtTokenDeposits - _amount;
    totalDebtTokenDeposits = newTotalDebtTokenDeposits;
    emit StabilityPoolDebtBalanceUpdated(newTotalDebtTokenDeposits);
```

Recommendations:

Additionally, burn the amount.

Yala: Resolved at @TroveManager.sol#L365...

Zenith: Verified

[L-5] Incorrect event emission when withdrawing

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- StabilityPool.sol

Description:

The withdraw event emits the original value of `_amount` rather than the capped value in case the amount was actually capped

```
function withdrawFromSP(uint256 _amount) external nonReentrant {

        ....

    uint256 debtToWithdraw = YalaMath._min(_amount, compoundedDebtDeposit);

        ...

    emit Withdraw(msg.sender, newDeposit, _amount);
}
```

Recommendations:

Emit `debtToWithdraw` instead

Yala: Resolved at StabilityPool.sol#L133

Zenith: Verified.

[L-6] BridgeController always assumes e18 decimals for delay classification

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BridgeController.sol
- BridgeController.sol

Description:

The `getDelay` assumes e18 as the decimal always in-order to the group the token amounts

```
function getDelay(IBridgeToken token, uint256 amount)
    public view override returns (uint256 delay) {
    if (amount / 1e18 == 0) {
        return tokenConfigs[token].refundDelays[RefundDelayType.LOW];
    }
    if (amount / 10e18 == 0) {
        return tokenConfigs[token].refundDelays[RefundDelayType.MIDDLE];
    }
    return tokenConfigs[token].refundDelays[RefundDelayType.HIGH];
}
```

This is flawed as tokens are created with custom decimals in `createBridgeToken`

```
function createBridgeToken(string memory name, string memory symbol,
    uint8 decimals, uint256 cap, address admin, bytes32 salt)
    external override onlyRole(DEFAULT_ADMIN_ROLE)
    returns (IBridgeToken token) {
    token = new BridgeToken{ salt: salt }(name, symbol, decimals, cap,
    admin);
    tokenConfigs[token].exists = true;
    emit BridgeTokenCreated(token);
}
```

Eg: A token is created with e6 decimals. Now all realistic token amounts will fall under the

LOW grouping

Recommendations:

If custom token decimals are planned to be created, use the tokens decimals inorder to categorize rather than hardcoding e18

Yala: Resolved at BridgeController.sol#L58-L67

Zenith: Verified

[L-7] Negative values for prices are currently allowed

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- PriceFeed.sol

Description:

When validating the price it is checked that `_response.answer` is not equal to 0. Since `_response.answer` is of int type this check still allows for negative values

```
function _isValidResponse(FeedResponse memory _response)
    internal view returns (bool) {
        return (_response.success) && (_response.roundId != 0) &&
    (_response.timestamp != 0) && (_response.timestamp <= block.timestamp)
    && (_response.answer != 0);
    }
```

Recommendations:

Since -ve values doesn't make sense for the application in scope, perform `> 0` validation instead

Yala: Resolved with @4dcc36bee1...

Zenith: Verified.

[L-8] TCR restrictions on withdrawals can cause dependence on other positions

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BorrowerOperations.sol

Description:

Withdrawals are only allowed if the totalCR of the TroveManager will be greater than CCR after the withdrawal

```
function closeTrove(ITroveManager troveManager, uint256 id,
address receiver) external auth(troveManager, id) {

    ....

    if (!troveManager.hasShutdown()) {
        uint256 newTCR = _getNewTCRFromTroveChange(vars.totalCollateral
* vars.price, vars.totalDebt + vars.totalInterest, coll * vars.price,
false, vars.compositeDebt, false);
=>  _requireNewTCRisAboveCCR(newTCR, vars.CCR);
    }
```

Since each individual position's liquidation ratio (MCR) is lower than CCR, this creates a scenario of dependence where one trove can be prevented from closing by another trove

Eg: CCR = 1.5 MCR = 1.1

Trove A is created with collateral == 200 and debt == 100 (assume collateral price == 1 and hence CR == 2) Another Trove B is opened with collateral = 120 and debt == 100

Now Trove A cannot be closed because doing so will drop the TCR to 1.2. Nor can B be liquidated because its CR is above MCR

Recommendations:

Acknowledge

Yala: We admit this issue, we may put a deposit when the TCR number is not healthy enough to unlock this.

Zenith: Acknowledged

[L-9] Debt repayments cannot be made in case protocol is paused

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BorrowerOperations.sol

Description:

In case the protocol is paused, then the `adjustTroves` function attempts to still allow pure repayments of debt. But the underlying `_adjustTroves` will disable this due to it reverting if the protocol is Paused without handling/specially considering the `just debt repayment` scenario

```
    function adjustTrove(ITroveManager troveManager, uint256 id,
    uint256 _collDeposit, uint256 _collWithdrawal, uint256 _debtChange,
    bool _isDebtIncrease) external {
=>  require((_collDeposit == 0 && !_isDebtIncrease) || !YALA_CORE.paused(),
    "BorrowerOps: Trove adjustments are paused");
        require(_collDeposit == 0 || _collWithdrawal == 0, "BorrowerOps:
    Cannot withdraw and add coll");
        _adjustTrove(troveManager, id, _collDeposit, _collWithdrawal,
    _debtChange, _isDebtIncrease);
    }


    function _adjustTrove(ITroveManager troveManager, uint256 id,
    uint256 _collDeposit, uint256 _collWithdrawal, uint256 _debtChange,
    bool _isDebtIncrease) internal {
=>  require(!YALA_CORE.paused(), "BorrowerOps: Trove adjustments are
    paused");
```

Recommendations:

Be consistent across both pause controls

Yala: Resolved with @5a9ce0a0f8...

Zenith: Verified

[L-10] Short circuiting interest rewards is flawed

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- TroveManager.sol

Description:

In case the collateral and debt difference b/w the snapshot and the current values of the TroveManager is 0, then the `getPendingRewards` function assumes that the defaulted interest to accrue is also 0

```
function getPendingRewards(uint256 id) public view returns (uint256,
    uint256, uint256) {
    RewardSnapshot memory snapshot = rewardSnapshots[id];
    uint256 coll = L_collateral - snapshot.collateral;
    uint256 debt = L_debt - snapshot.debt;
    uint256 defaulted = L_defaulted_interest - snapshot.defaultedInterest;
    if (coll + debt == 0 || !_exists(id)) return (0, 0, 0);
```

But this is flawed as there can be non-zero interest defaulted while `L_collateral` and `L_debt` doesn't get incremented

POC: price == 1e(5 + 28) (coin == wbtc and hence price decimals == 36 - 8) penaltyRatio == 0.1e18 singleLiquidation.debtRedistOrJIT == 0 (fully liquidated by Stability pool) singleLiquidation.interestRedistOrJIT == 0.8e15 now collateral seized == (0.8e15 * (1e18 + 0.1e18)) / 1e33 == 0

Hence `L_debt` and `L_collateral` while `L_interest` will be incremented. This causes this amount to not be passed on to the troves (those that were opened right before this liquidation happens). And this defaulted interest will remain as bad debt forever

Recommendations:

Don't short circuit early and always check for delta b/w snapshot interest and the current interest value

Yala: Resolved on TroveManager.sol#L288-L289

Zenith: Verified

[L-11] `maxSeizedColl` value is rounded down in favor of defaulters

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- TroveManager.sol

Description:

The `maxSeizedColl` value is rounded down. This favors the defaulter instead of the good behaving troves and will cause the trove to accrued debt with lower compensation given

```
uint256 maxSeizedColl = (_debtToLiquidate * (DECIMAL_PRECISION + _penaltyRatio))
/ _price;
```

POC:

POC: price == 1e(5 + 28) (coin == wbtc and hence price decimals == 36 - 8) penaltyRatio == 0.1e18 debtToLiquidate == 0.8e15 maxSeizedColl == 0

This will cause the troves to accrue more debt for 0 compensation given

Recommendations:

Round up in favor of the troves instead of the defaulter

Yala: Acknowledged.

[L-12] Defaulted debt and interest is not updated in case collateral value remains same

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- TroveManager.sol
- TroveManager.sol

Description:

The defaulted debt and interest is only passed on to the borrowers in case their `collateral snapshot` is less than `L_collateral`

```
function applyPendingRewards(uint256 id) public returns (uint256 coll,
    uint256 debt, uint256 interest) {
    Trove storage t = Troves[id];
    if (_exists(id)) {
        debt = t.debt;
        coll = t.coll;
        (interest, ) = _accrueTroveInterest(id);
        if (rewardSnapshots[id].collateral < L_collateral) {
            // Compute pending rewards
            (uint256 pendingCollateralReward, uint256 pendingDebtReward,
    uint256 pendingDefaultedInterest) = getPendingRewards(id);

              ....
        }

        _updateTroveRewardSnapshots(id);
```

This is flawed as defaulted debt and defaulted interest can exist even when `L_collateral` remains same:

```
uint256 maxSeizedColl = (_debtToLiquidate * (DECIMAL_PRECISION
    + _penaltyRatio)) / _price;
```

POC: price == 1e(5 + 28) (coin == wbtc and hence price decimals == 36 - 8) penaltyRatio

== 0.1e18 debtToLiquidate == 0.8e15 hence maxSeizedColl == 0

this will cause the `L_interest` variable to remain the same (unless there was some high error in previous calculation) while `L_debt` and `L_defaulted_interest` will be incremented

Since this defaulted debt and interest will not be passed on to the troves, it will remain as bad debt to the protocol forever

Recommendations:

Compare corresponding snapshot with `L_debt` and `L_defaulted_interest` instead of returning early

Yala: Resolved with @ab624a09ed...

Zenith: Verified. Now all three are compared instead of just the collateral snapshot

4.5 Informational

A total of 2 informational findings were identified.

[I-1] computeNewStake always return 1:1 ratio

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- TroveManager.sol

Description:

The `totalCollateralSnapshot` and `totalStakesSnapshot` variables are never updated. Hence the `_computeNewStake` function always return 1:1 ratio

```
function _computeNewStake(uint256 _coll) internal view returns (uint256) {
    uint256 stake;
    if (totalCollateralSnapshot == 0) {
        stake = _coll;
    } else {
        /*
         * The following assert() holds true because:
         * - The system always contains ≥ 1 trove
         * - When we close or liquidate a trove, we redistribute the pending
    rewards, so if all troves were closed/liquidated,
         * rewards would've been emptied and totalCollateralSnapshot would be
    zero too.
         */
        stake = (_coll * totalStakesSnapshot) / totalCollateralSnapshot;
    }
    return stake;
}
```

Recommendations:

Yala: Resolved with @3440fd8ab1c... and @9d692df85ef...

Zenith: Verified

[I-2] User can bypass `buy` and `sell` fee in `PSM`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- PSM.sol#L86
- PSM.sol#L91

Description:

When users call `buy` or `sell`, the `estimateBuy` and `estimateSell` functions calculates the fee they need to pay.

```
    function estimateBuy(uint256 amountDebtToken)
    public view override returns (uint256 amountPegTokenUsed, uint256 fee) {
>>>   fee = (amountDebtToken * feeIn) / DECIMAL_PRECISIONS;
        amountPegTokenUsed = (amountDebtToken + fee) / priceFactor;
    }

    function estimateSell(uint256 amountDebtToken)
    public view override returns (uint256 amountPegTokenReceived,
    uint256 fee) {
>>>   fee = (amountDebtToken * feeOut) / DECIMAL_PRECISIONS;
        amountPegTokenReceived = (amountDebtToken - fee) / priceFactor;
    }
```

However, due to the lack of a minimum `amountDebtToken`, users can provide an `amountDebtToken` that results in a `fee` of 0.

Recommendations:

Consider checking the minimum `amountDebtToken` or reverting when the configured `feeIn`/`feeOut` is non-zero but the calculated `fee` results in 0.

Yala: Resolved with @e5d794ced1...

Zenith: Verified.