Ouroboros Security Review

Pashov Audit Group

Conducted by: btk, samuraii77, Shaka

December 6th 2024 - December 26th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Ouroboros-Protocol/Ouroboros-USDx** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Ouroboros

Ouroboros is a borrowing and CDP stablecoin protocol with two main tokens - USDx and ORX. USDx is a fully collateralized, dollar-pegged stablecoin that uses assets like TitanX and DragonX as collateral. ORX is a limited-supply ERC20 token that facilitates staking rewards and fee distribution within the Ouroboros ecosystem. It is allocated through a vesting mechanism for TitanX deposits over 52 weeks and a linear dripping process for ETH contributions over 12 weeks.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 4801614104f3fc8f1e3fd465322455010ff1ed90

fixes review commit hash - d5f529ad4ec39d87e48a2a77a0898cee5e2c8ab2

Scope

The following smart contracts were in scope of the audit:

- `BackstopPool`
- `BaseRateAbstract`
- `GasCompensationPool`
- `PositionController`
- `Stable`
- `NonPayableProxy`
- `CollateralControllerState`
- `CollateralControllerImpl`
- `CollateralController`
- `ActivePool`
- `CollateralSurplusPool`
- `DefaultPool`
- `LiquidationManager`
- `PositionManager`
- `PositionState`
- `SortedPositions`
- `DragonXPriceFeed`
- `EthPriceFeed`
- `TitanXPriceFeed`
- `HintHelpers`
- `MultiTroveGetter`

7. Executive Summary

Over the course of the security review, btk, samuraii77, Shaka engaged with Ouroboros to review Ouroboros. In this period of time a total of **22** issues were uncovered.

Protocol Summary

Protocol Name	Ouroboros
Repository	https://github.com/Ouroboros-Protocol/Ouroboros-USDx
Date	December 6th 2024 - December 26th 2024
Protocol Type	CDP Stablecoin

Findings Count

Severity	**Amount**
Medium	11
Low	11
Total Findings	**22**

Summary of Findings

ID	Title	Severity	Status
[M-01]	TCR is overestimated on redemptions	Medium	Acknowledged
[M-02]	DoS due to Chainlink oracle returning wrong data or going offline	Medium	Acknowledged
[M-03]	getRedemptionHints will revert if _maxIterations is reached	Medium	Resolved
[M-04]	Max price deviation is too high	Medium	Resolved
[M-05]	Current redemption design causes users to be charged a penalty despite doing nothing wrong	Medium	Acknowledged
[M-06]	Uniswap oracle prices can be manipulated	Medium	Resolved
[M-07]	Base rate may decay at a slower rate than expected	Medium	Resolved
[M-08]	Users with escrowed stables are charged a penalty if TCR < MCR during the grace period	Medium	Resolved
[M-09]	Queued stables can get stuck in PositionManager after a collateral is sunset	Medium	Resolved
[M-10]	Borrower will unfairly lose his stables if the collateral goes into recovery mode during sunset period	Medium	Acknowledged
[M-11]	DoS in redemptions due to inability to consume reescrowed stables	Medium	Acknowledged
[L-01]	adjustPosition does not allow repaying stable in the decommissioning period	Low	Resolved

[L-02]	Inconsistent behavior of fetchEthPrice() when Chainlink oracle fails	Low	Acknowledged
[L-03]	Misleading calculation of price deviation	Low	Resolved
[L-04]	getGlobalStats always returns zero address for guardian	Low	Resolved
[L-05]	Not using SafeTransferFrom	Low	Resolved
[L-06]	Wrong signs cause an incorrect revert	Low	Resolved
[L-07]	Rate limits can prevent the system recovery via debt issuance	Low	Acknowledged
[L-08]	Users can pay less fees by abusing the utilization ratio fee calculation	Low	Acknowledged
[L-09]	Loss of precision in loan and redemption point regeneration slows the regeneration rate	Low	Resolved
[L-10]	Calls to Chainlink do not prevent griefing attacks	Low	Resolved
[L-11]	Burning clawback rewards causes total debt to be greater than stable total supply	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] TCR is overestimated on redemptions

Severity

Impact: Medium

Likelihood: Medium

Description

The system uses different price types (lowest, highest, and weighted average) depending on the nature of the operation. As such, in some operations, the most conservative price from the point of view of the protocol safety is used.

For example, using the lowest price for debt creation makes sure that the amount of debt created is not higher than it should be.

In the same way, using the highest price for redemptions makes sure that the amount of collateral withdrawn from the protocol is not higher than it should be. However, on `redeemCollateral` this same price is also used to check if the TCR is above the MCR. So using the highest price can overestimate the TCR and, consequently, allow redemptions when the real TCR is below the MCR.

```
File: PositionManager.sol

423:  @>     (
  totals.price,
  totals.suggestedAdditiveFeePCT
) = priceFeed.fetchHighestPriceWithFeeSuggestion(
424:              totals.loadIncrease,
425:              totals.utilizationPCT,
426:              true,
427:              true
428:          );
429:
430:          _requireValidMaxFeePercentage
  (totals.maxFeePCT, totals.suggestedAdditiveFeePCT);
431:
432:          uint MCR = collateralController.getMCR(address
  (collateralToken), totals.version);
433:  @>     _requireTCRoverMCR(totals.price, MCR);
```

Recommendations

Use the lowest or weighted average price to calculate the TCR.

[M-02] DoS due to Chainlink oracle returning wrong data or going offline

Severity

Impact: High

Likelihood: Low

Description

Chainlink oracle price feed can return wrong data or go offline in extreme cases, which will provoke the `EthPriceFeed.fetchEthPrice()` function to revert. This function is called to execute the main actions of the system, such as redeeming collateral, opening a position, adding collateral, closing a position, claiming escrow, and liquidating a position.

The impact of transaction reverting is especially severe in the case of liquidations, which might cause the system to be uncollateralized, and adding collateral, which might cause users to lose their collateral when the price feed gets back online.

In order to prevent this, the system relies on the guardian calling the `resetBreaker` function to manually provide the new price and set the oracle status to the working state. However, this is not ideal, as:

- Requires the guardian to act quickly after the oracle goes offline.
- Until the oracle goes back online, the process has to be repeated after every interaction with the system, as every call will change the oracle status again to not working.
- Introduces a centralization vector.

Another issue is that price movements greater than 50% are considered invalid, which prevents the system from reacting quickly to extreme price movements.

Recommendations

Use a fallback oracle to query the price in case the main oracle fails or the price movement is too high.

[M-03] `getRedemptionHints` will revert if `_maxIterations` is reached

Severity

Impact: Low

Likelihood: High

Description

The function `getRedemptionHints()` in `HintHelpers.sol` is used to find the right hints to pass to `redeemCollateral()`. The number of positions to consider for redemption can be capped to `_maxIterations` to avoid running out of gas. However, there is a flaw in the implementation that will cause the function to revert if `_maxIterations` is reached.

```
File: HintHelpers.sol

48:         while (vars.currentPositionUser != address
  (0) && vars.remainingStables > 0 && _maxIterations-- > 0) {
```

The decrement of the `_maxIterations` variable is done after the condition check, meaning that in the last iteration, when `_maxIterations` is 0, the condition evaluates to false, but the decrement is still executed, causing an underflow and the function to revert.

The original code from Liquity is not subject to this issue, as it intentionally allows the underflow to happen. However, using solc 0.8.0 or higher, the code should be adapted to avoid reverting due to the underflow.

The result is that queries that entail a large number of iterations (i.e., the loop is not exited before `_maxIterations` is reached) will revert.

Recommendations

```
-       while (vars.currentPositionUser != address
- (0) && vars.remainingStables > 0 && _maxIterations-- > 0) {
+       uint i;
+       while (vars.currentPositionUser != address
+ (0) && vars.remainingStables > 0 && i++ < _maxIterations) {
```

[M-04] Max price deviation is too high

Severity

Impact: High

Likelihood: Low

Description

The `MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND` parameter is intended to safeguard against price manipulation. It is utilized within the `_chainlinkPriceChangeAboveMax()` function to verify if the price has undergone a substantial change between consecutive rounds:

```
function _chainlinkPriceChangeAboveMax(
        ChainlinkResponse memory _currentResponse,
        ChainlinkResponse memory _prevResponse
    ) internal pure returns (bool) {
        // code ...

        // Return true if price has more than doubled, or more than halved.
        return percentDeviation > MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND;
    }
```

The function returns true only if the price has more than doubled or more than halved. This logic implies that if the price changes by 49%, the function will deem it valid. However, such a significant price shift between rounds due to normal market conditions is highly unlikely. Consequently, the current configuration allows for a potential 49% price manipulation window, which is too high.

Recommendations

It is recommended to lower the `MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND`. A more conservative value, such as 20%, would align with best practices seen in protocols like FortressFinance.

[M-05] Current redemption design causes users to be charged a penalty despite doing nothing wrong

Severity

Impact: Medium

Likelihood: Medium

Description

Upon users redeeming their escrow positions, we have the following piece of code if the cooldown and grace period have expired:

```
else {
    // We are about to return stables to user and charge the timeout fee on
    // total remaining amount
    timedOut = true;
    effectiveAmount = redemption.stableAmount;
    redemption.stableAmount = 0;
}
```

In this case, the user will be charged a penalty and his escrow position will be canceled. There are 2 scenarios for this to occur:

Scenario 1:

1. User creates an escrow position to redeem his debt tokens for collateral

2. He does not redeem his position on time as he is not a serious user
3. He is rightfully charged a penalty fee

Scenario 2:

1. User creates an escrow position
2. He tries to redeem his position over and over again however there are not enough positions to redeem collateral from, we then end up here (we could theoretically even revert before reaching the below code in the collateral drawn above 0 check as well):

```
else if (unusedStables != 0) {
            // otherwise, transfer back left over stables for next attempt
            stableToken.transferForRedemptionEscrow(msg.sender, address
              (this), unusedStables);
            escrow.stableAmount += unusedStables;
        }
```

3. In the above code, we fill up the escrow position again with the unused tokens
4. The grace period is over and the user is charged a fee despite not doing anything wrong

Recommendations

A possible fix is to add a boolean flag in the redemption position indicating whether the user has tried redeeming his collateral. Then, upon applying the penalty, if he has tried redeeming the collateral, only apply it if the boolean is false.

[M-06] Uniswap oracle prices can be manipulated

Severity

Impact: High

Likelihood: Low

Description

The protocol uses Uniswap V3 oracles to calculate the prices of the collateral tokens TitanX and DragonX. For the price of TitanX, the ETH/TITANX pool is used, while for the price of DragonX, both the ETH/TITANX and the TITANX/DRAGONX pools are used.

There are different factors that make the manipulation of these prices feasible:

- The current liquidity in the pools is relatively low: $4.7M in the ETH/TITANX pool and $1.9M in the TITANX/DRAGONX pool.
- The TWAP windows are too short: 36 seconds for the short TWAP and 5 minutes for the long TWAP.
- In the case of DragonX, using two oracles makes it easier to manipulate the price, as manipulating the price in both pools can amplify the effect on the final price.

The protocol uses some safeguards to prevent the price manipulation in some operations:

- Checks for price deviations higher than a certain threshold.
- Using different TWAP windows and choosing the most conservative price.

The first safeguard cannot be used for all operations or be too strict, as it could result in a denial of service for legitimate users. The second safeguard, while useful, is not enough to prevent price manipulation.

Significant changes in the price of DragonX can be achieved with relatively low capital, especially in the case of the weighted average price, and when the attacker is assigned to propose 3 blocks in a row. While the latter might seem very unlikely, it has been estimated that a validator with a 1% share is expected to be assigned to 3 blocks in a row on average once every 5 months.

An attacker could exploit this vulnerability to profit from the price manipulation, either by liquidating positions, claiming escrowed funds, or closing existing positions.

Proof of Concept

This proof of concept was created with Foundry, so it is necessary to add it to the project.

To run the test in a fork of the Ethereum mainnet, use the following command, replacing `{FORK_URL}` with a RPC URL:

```
forge test -vv --mt test_manipulateTWAP --fork-url {FORK_URL}`
```

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import { IERC20 } from "forge-std/interfaces/IERC20.sol";
import "contracts/stable/collateral/oracle/TitanXPriceFeed.sol";
import "contracts/stable/collateral/oracle/DragonXPriceFeed.sol";

contract AuditOracleTests is Test {
    address UNI_FACTORY = 0x1F98431c8aD98523631AE4a59f267346ea31F984;
    address SWAP_ROUTER = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
    address PRICE_AGGREGATOR = 0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419;
    address WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address TITANX = 0xF19308F923582A6f7c465e5CE7a9Dc1BEC6665B1;
    address DRAGONX = 0x96a5399D07896f757Bd4c6eF56461F58DB951862;

    uint32 shortTwapSeconds = 36 seconds;
    uint32 longTwapSeconds = 5 minutes;

    TitanXPriceFeed titanXPriceFeed;
    DragonXPriceFeed dragonXPriceFeed;

    uint256 prevDragonUsdSpotPrice;
    uint256 prevDragonUsdShortTWAP;
    uint256 prevDragonUsdLongTWAP;
    uint256 prevDragonUsdLowestPrice;
    uint256 prevDragonUsdWeightedAveragePrice;

    function setUp() public {
        address[] memory _priceAggregatorAddresses = new address[](1);
        _priceAggregatorAddresses[0] = PRICE_AGGREGATOR;

        titanXPriceFeed = new TitanXPriceFeed(
            UNI_FACTORY,
            TITANX,
            WETH,
            address(this),
            address(this),
            _priceAggregatorAddresses
        );

        dragonXPriceFeed = new DragonXPriceFeed(
            UNI_FACTORY,
            DRAGONX,
            TITANX,
            WETH,
            address(this),
            address(this),
            _priceAggregatorAddresses
        );
        dragonXPriceFeed.setPositionManager(address(this));
    }

    function test_manipulateTWAP() public {
        uint256 wethIn = 100e18;
        deal(WETH, address(this), wethIn);

        _skip(1);
        _logPrices();

        uint256 titanOut = _swap(wethIn, WETH, TITANX);
        uint256 dragonOut = _swap(titanOut, TITANX, DRAGONX);

        _skip(1);
        _logPrices();

        titanOut = _swap(dragonOut, DRAGONX, TITANX);
```

```
        uint256 wethOut = _swap(titanOut, TITANX, WETH);

        uint256 wethCost = wethIn - wethOut;
        uint256 ethPrice = titanXPriceFeed.fetchEthPrice();
        uint256 usdInitialCapital = wethIn * ethPrice / 1e18;
        uint256 usdCost = wethCost * ethPrice / 1e18;
        console.log("Initial capital: %s WETH (%s USD)", _toDec
          (wethIn), usdInitialCapital / 1e18);
        console.log("Cost: %s WETH (%s USD)", _toDec(wethCost), usdCost / 1e18);
    }

    function _swap(
      uint256amountIn,
      addresstokenIn,
      addresstokenOut
    ) public returns (uint256
        IERC20(tokenIn).approve(SWAP_ROUTER, amountIn);

        uint256 amountOut = ISwapRouter(SWAP_ROUTER).exactInputSingle(
            ISwapRouter.ExactInputSingleParams({
                tokenIn: tokenIn,
                tokenOut: tokenOut,
                fee: 10_000,
                recipient: address(this),
                deadline: block.timestamp,
                amountIn: amountIn,
                amountOutMinimum: 0,
                sqrtPriceLimitX96: 0
            })
        );

        return amountOut;
    }

    function _skip(uint256 blocks) private {
        vm.roll(block.number + blocks);
        vm.warp(block.timestamp + blocks * 12);
    }

    function _logPrices() private {
        uint256 dragonUsdSpotPrice = dragonXPriceFeed.getPrice(0);
        uint256 dragonUsdShortTWAP = dragonXPriceFeed.getPrice
          (shortTwapSeconds);
        uint256 dragonUsdLongTWAP = dragonXPriceFeed.getPrice(longTwapSeconds);
        uint256 dragonUsdLowestPrice = dragonXPriceFeed.fetchLowestPrice
          (false, false);


        string memory dragonUsdSpotPriceChange;
        string memory dragonUsdShortTWAPChange;
        string memory dragonUsdLongTWAPChange;
        string memory dragonUsdLowestPriceChange;
        string memory dragonUsdWeightedAveragePriceChange;

        dragonUsdSpotPriceChange = _percentChangeString
          (prevDragonUsdSpotPrice, dragonUsdSpotPrice);
        dragonUsdShortTWAPChange = _percentChangeString
          (prevDragonUsdShortTWAP, dragonUsdShortTWAP);
        dragonUsdLongTWAPChange = _percentChangeString
          (prevDragonUsdLongTWAP, dragonUsdLongTWAP);
        dragonUsdLowestPriceChange = _percentChangeString
          (prevDragonUsdLowestPrice, dragonUsdLowestPrice);
        dragonUsdWeightedAveragePriceChange = _percentChangeString
          (prevDragonUsdWeightedAveragePrice, dragonUsdWeightedAveragePrice);

        prevDragonUsdSpotPrice = dragonUsdSpotPrice;
        prevDragonUsdShortTWAP = dragonUsdShortTWAP;
        prevDragonUsdLongTWAP = dragonUsdLongTWAP;
```

```
        prevDragonUsdLowestPrice = dragonUsdLowestPrice;
        prevDragonUsdWeightedAveragePrice = dragonUsdWeightedAveragePrice;

        console.log("=> DragonX price (in USD)");
        console.log
          ("spot:         %e", dragonUsdSpotPrice, dragonUsdSpotPriceChange);
        console.log
          ("short TWAP:   %e", dragonUsdShortTWAP, dragonUsdShortTWAPChange);
        console.log
          ("long TWAP:    %e", dragonUsdLongTWAP, dragonUsdLongTWAPChange);
        console.log(
          "lowest:%e",
          dragonUsdLowestPrice,
          dragonUsdLowestPriceChange
        );
        console.log(
          "weightedavg:%e",
          dragonUsdWeightedAveragePrice,
          dragonUsdWeightedAveragePriceChange
        );
        console.log("");
    }

    function _percentChangeString
      (uint256 prev, uint256 current) private pure returns (string memory) {
        if (prev == 0) return "";
        int256 change = int256(current) - int256(prev);
        int256 percentChange = (change * 100) / int256(prev);
        if (percentChange == 0) return "";
        return string.concat("(", percentChange > 0 ? "+" : "", vm.toString
          (percentChange), "%)");
    }

    function _toDec(uint256 value) private returns (string memory) {
        uint256 integerPart = value / 1e18;
        uint256 fractionalPart = (value % 1e18) / 1e16;
        return string.concat(vm.toString(integerPart), ".", vm.toString
          (fractionalPart));
    }
}
```

Console output:

```
=> DragonX price (in USD)
  spot:         6.217954575461e12
  short TWAP:   6.216051602698e12
  long TWAP:    6.216051610562e12
  lowest:       6.216051602698e12
  weighted avg: 6.21605160794e12

  => DragonX price (in USD)
  spot:         8.508795576428e12 (+36%)
  short TWAP:   6.902127347209e12 (+11%)
  long TWAP:    6.296124413169e12 (+1%)
  lowest:       6.296124413169e12 (+1%)
  weighted avg: 6.498125391182e12 (+4%)

  Initial capital: 100.0 WETH (393203 USD)
  Cost: 3.63 WETH (14302 USD)
```

Recommendations

Increase the TWAP windows to increase the resistance to price manipulation. This change might also require increasing the observation cardinality for the ETH/TITANX and TITANX/DRAGONX pools.

[M-07] Base rate may decay at a slower rate than expected

Severity

Impact: Medium

Likelihood: Medium

Description

The base rate is used to calculate fees for redemptions and new issuances of debt, and its value decays over time and increases based on redemption volume.

On redemption, the decay on the base rate is calculated based on the number of minutes passed since the last fee operation. What is important to note is that the number of minutes passed is rounded down to the nearest minute.

```
File: BaseRateAbstract.sol
68:     function _minutesPassedSinceLastFeeOp() internal view returns (uint) {
69:         return
  (block.timestamp - _lastFeeOperationTime) / SECONDS_IN_ONE_MINUTE;
70:     }
```

However, when `_lastFeeOperationTime` is updated at the end of the operation, the value is stored in the current block timestamp.

```
File: BaseRateAbstract.sol
80:     _lastFeeOperationTime = block.timestamp;
```

This inconsistency can lead to the base rate decaying slower than expected.

For example, if 119 seconds have passed since the last fee operation, the number of minutes passed rounds down to 1. The base rate will decay as if only 60 seconds had passed, however, `_lastFeeOperationTime` will be updated taking into account the full 119 seconds. This means that in the worst-case

scenario, the base rate will only decay for the amount corresponding to 60 seconds every 119 seconds.

Recommendations

```
function _updateLastFeeOpTime() internal {
-       uint timePassed = block.timestamp - _lastFeeOperationTime;
+       uint minutesPassed = _minutesPassedSinceLastFeeOp();

-       if (timePassed >= SECONDS_IN_ONE_MINUTE) {
+       if (minutesPassed > 0) {
-           _lastFeeOperationTime = block.timestamp;
+           _lastFeeOperationTime += minutesPassed * SECONDS_IN_ONE_MINUTE;
            emit LastFeeOpTimeUpdated(block.timestamp);
        }
    }
```

[M-08] Users with escrowed stables are charged a penalty if TCR < MCR during the grace period

Severity

Impact: Medium

Likelihood: Medium

Description

When the redemption cooldown requirement is set, users need to escrow their stables for a certain period before being able to redeem collateral. After the cooldown period, the grace period starts, and users can redeem their collateral. However, if the redemption does not take place before the grace period ends, the user will be charged a penalty fee.

Given that the `redeemCollateral` function reverts if the TCR is below the MCR, if the TCR falls below the MCR during the cooldown period and does not recover before the grace period ends, users will be unable to redeem their collateral and will be charged a penalty.

```
File: PositionManager.sol

433:        _requireTCRoverMCR(totals.price, MCR);
```

Recommendations

Allow users to dequeue their escrowed stables when TCR < MCR.

[M-09] Queued stables can get stuck in `PositionManager` after a collateral is sunset

Severity

Impact: High

Likelihood: Low

Description

After a collateral is sunset, all the orphaned collateral tokens can be withdrawn by the guardian. If users have queued redemptions, their queued stables will get stuck in the `PositionManager` contract, as there will not be any collateral to redeem. One way of allowing users to redeem their stables it setting `redemptionCooldownPeriod` to 0, as this will trigger `_dequeueEscrow()` when `redeemCollateral()` is called. However, this value cannot be set on sunset collaterals.

Recommendations

Set `redemptionCooldownPeriod` to 0 on `activeToSunset()` to allow users to recover their escrowed stables after the collateral is sunset.

[M-10] Borrower will unfairly lose his stables if the collateral goes into recovery mode during sunset period

Severity

Impact: Medium

Likelihood: Medium

Description

The PositionController contract utilizes an escrow mechanism to temporarily hold a borrower's stablecoins before they can be claimed. However, an edge case exists where the borrower may lose both their collateral and stablecoins. For instance, consider the following scenario:

- A borrower opens a position, and their stablecoins are placed in escrow.
- Shortly thereafter, the collateral is sunset (e.g., due to being highly volatile).
- The price of the collateral drops, pushing the system into recovery mode.

After the escrow period ends, the borrower would be unable to claim their stablecoins or their collateral. Furthermore, they would not be able to add additional collateral to improve (ICR) and withdraw their funds because the collateral has been decommissioned.

Recommendations

Introduce a functionality that allows escrowed users to add more collateral during the decommissioning period to withdraw their stables.

[M-11] DoS in redemptions due to inability to consume reescrowed stables

Severity

Impact: Medium

Likelihood: Medium

Description

On collateral redemptions, it is possible that not all the `_stableAmount` is consumed if the remaining debt of the position used for redemption is lower than the minimum net debt and, when the cooldown redemption requirement is in place, the remaining stables are re-escrowed.

```
File: PositionManager.sol

288:         if (
289:             ((newDebt - GAS_COMPENSATION) < MIN_NET_DEBT) ||
290:             ((newNICR != _partialRedemptionHintNICR) && !firstInLine)
291:         ) {
292:             singleRedemption.cancelledPartial = true;
293:             return singleRedemption;
294:         }
    (...)
483:         if (totals.cooldownRequirement != 0) {
484:             _checkIfEscrowDepletedAndReEscrowUnusedStables
  (totals.remainingStables);
485:         }
```

In order to consume the remaining escrowed stables, the user has to call `redeemCollateral` again.

However, it is possible that the transaction will revert due to different causes:

1. The most likely scenario is that the position to redeem from is the same that was not possible to redeem from in the first place, due to the same conditions, causing the transaction to revert due to the lack of collateral to redeem from.
2. If the re-escrowed stable amount is too low, the transaction can revert on the update of the base rate. This can happen if a) the current base rate is zero and b) `(_CollateralDrawn * _price) / _totalStableSupply` rounds down to zero.

```
File: BaseRateAbstract.sol

33:     function _updateBaseRateFromRedemption(
  uint_CollateralDrawn,
  uint_price,
  uint_totalStableSupply
) internal returns (uint
34:         // First, decay the base rate based on time passed since last fee
// operation
35:         uint decayedBaseRate = _calcDecayedBaseRate();
36:
37:         // Calculate the fraction of total supply that was redeemed at face
// value
38:         uint redeemedStableFraction =
  (_CollateralDrawn * _price) / _totalStableSupply;
39:
40:         // Increase the base rate based on the redeemed fraction
41:         uint newBaseRate = decayedBaseRate + (redeemedStableFraction / BETA);
42:         newBaseRate = StableMath._min
//(newBaseRate, StableMath.DECIMAL_PRECISION); // Cap base rate at 100%
43:         assert
//(newBaseRate > 0); // Base rate is always non-zero after redemption
```

This will cause that the user cannot complete the redemption and, thus, queue a new redemption, until the grace period expires. Additionally, the user will be

charged the timeout fee.

Recommendations

Consider dequeuing the remaining sables when the user tries to consume all the escrowed stables but the current conditions do not allow it.

8.2. Low Findings

[L-01] `adjustPosition` does not allow repaying stable in the decommissioning period

The `adjustPosition` function in `PositionController.sol` provides users with the flexibility of adjusting their positions by increasing or decreasing their collateral and/or stable. However, the function does not allow them to repay their stable in the decommissioning period, while this is allowed through the `repayStable` function.

As a result, users that want to repay their stable and withdraw their collateral in the decommissioned period are forced to perform two calls, one to `repayStable` and another to `withdrawColl`, instead of being able to use the `adjustPosition` function.

Consider modifying the `adjustPosition` function to allow repaying stable in the decommissioning period.

```
--      if (_collAddition > 0 || _stableChange > 0) {_requireNotSunsetting
- (asset, version);}
++      if (_collAddition > 0 ||
+ (_stableChange > 0 && _isDebtIncrease)) {_requireNotSunsetting(asset, version);}
```

[L-02] Inconsistent behavior of `fetchEthPrice()` when Chainlink oracle fails

The `EthPriceFeed.fetchEthPrice()` function reverts when `status == Status.chainlinkBroken`. This status is set after the Chainlink oracle has returned wrong or outdated data, or the price has deviated too much from the previous price. However, when this happens, the latest valid price is returned.

This provokes that the wrong value can be returned in the transaction where the Chainlink oracle fails, but instead a revert happens in the next transaction.

Let's consider the following scenario: The current ETH price is $4000.

- Price falls to $1999.
- When `fetchEthPrice` is called it returns the latest valid price of $4000 and sets the status to `chainlinkBroken`.
- In the next transaction the price is still $1999, but in this case, the function reverts.

Consider either always reverting or always returning the latest valid price when the Chainlink oracle fails.

[L-03] Misleading calculation of price deviation

The `_priceDeviationInSafeRange()` function in `TitanXPriceFeed.sol` checks if the price deviation between the minimum and maximum price is within a safe range.

```
File: TitanXPriceFeed.sol

334:         uint avgPrice = (minPrice + maxPrice) / 2;
335:         uint percentDifference = (
  (maxPrice - minPrice) * DECIMAL_PRECISION) / avgPrice;
```

As we can see, for the calculation of the percentage difference it uses the average price as the base price. This calculation is misleading as it does not represent the actual percentage difference between the minimum and maximum price.

Let's consider the following example:

```
maxPrice = 100
minPrice = 10
percentDifference = (100 - 10) / ((100 + 10) / 2) = 90 / 55 = 1.63 = 163%
```

The result is 163%, which does not represent the actual percentage difference between the minimum and maximum price. This makes it difficult to interpret and choose the value of the `maxOracleDeltaPCT` parameter.

We should choose the minimum price as the basis for the calculation and interpret that the price has increased by 900%, or use the maximum price as the basis and interpret that the price has decreased by 90%.

In that regard, it is suggested to use the maxPrice as the base price, as it is consistent with the calculation of the percentage deviation in the `EthPriceFeed.sol` contract.

```
File: EthPriceFeed.sol

254:         uint percentDeviation = (
  (maxPrice - minPrice) * StableMath.DECIMAL_PRECISION) / maxPrice;
```

Consider applying the following changes to the code:

```
-         uint avgPrice = (minPrice + maxPrice) / 2;
-         uint percentDifference = (
- (maxPrice - minPrice) * DECIMAL_PRECISION) / avgPrice;
+         uint percentDifference = (
+ (maxPrice - minPrice) * DECIMAL_PRECISION) / maxPrice;
```

[L-04] `getGlobalStats` always returns zero address for guardian

The `getGlobalStats` function in `HintHelpers.sol` returns a struct that contains a `guardian` field. The data for this field is retrieved using the `getGuardian` function from the `CollateralController` contract.

```
File: HintHelpers.sol

206:     stats.guardian = collateralController.getGuardian();
```

As the `CollateralController` does not implement the `getGuardian` function itself, the call is delegated to the `CollateralControllerImpl` contract, which will return the value in the slot corresponding to the `_guardian` variable, which is the slot at index 18. However, in the `CollateralController` contract this slot corresponds to the `_roles` mapping of the `AccessControl` contract, which will always be empty. As the delegated call is executed in the context of the `CollateralController`, this means that `getGuardian` will always return the zero address.

It is recommended to change the order of the inheritance in the `CollateralController` contract to ensure the `_guardian` variable occupies the same slot as in the `CollateralControllerImpl` contract.

```
contract CollateralController
    is CollateralControllerState,
       NonPayableProxy,
-      TimelockController,
-      Guardable {
+      Guardable,
+      TimelockController {
```

[L-05] Not using `SafeTransferFrom`

The current implementation of `PostionController._activePoolAddColl()` does not support tokens that do not return a boolean on successful transfers. The issue lies in the following code:

```
require(
            collateral.asset.transferFrom(
                msg.sender,
                address(collateral.activePool),
                _amount
            ),
            "PositionController: Sending collateral to ActivePool failed"
        );
```

Consider using `safeTransferFrom()` to handle such tokens.

[L-06] Wrong signs cause an incorrect revert

In `PositionManager._validateAndReleaseStablesForRedemption()`, we have this check:

```
require(
  cooldownExpiry<block.timestamp,
  "Redemptionescrowtimelocknotsatisfied"
);
```

The check is wrong as it should be `<=` as that is the time when the cooldown expires. For example, if we take a look at this piece of code:

```
bool isGracePeriod = block.timestamp < gracePeriodExpiry;
```

We will see that it is considered grace period when the time is before `gracePeriodExpiry`, thus `gracePeriodExpiry` is considered to already be

expired. However, in the cooldown expiry check, the `cooldownExpiry` is considered non-expired.

[L-07] Rate limits can prevent the system recovery via debt issuance

When a position is opened the debt of an existing position is increased, and the system is in recovery mode, users are not charged any borrowing fees. This is done to maximally encourage borrowing, as the minimum required ICR is higher in recovery mode, and the issuance of new debt becomes a key mechanism to recover the system.

However, if all the available loan points are consumed, the system cannot rely anymore on the issuance of new debt to recover, and in extreme cases could lead to the system being undercollateralized.

Consider bypassing the consumption of loan points when the system is in recovery mode.

[L-08] Users can pay less fees by abusing the utilization ratio fee calculation

Opening a position incurs an additive fee based on the points utilization ratio:

```
(
  vars.price,
  vars.suggestedFeePCT
) = collateral.priceFeed.fetchLowestPriceWithFeeSuggestion(
            vars.loadIncrease,
            vars.utilizationPCT,
            true,
            true
        );
```

The utilization ratio is calculated using a typical formula:

```
return (usedPoints * DECIMAL_PRECISION) / maxPoints;
```

However, these 2 factors combined allow for users to pay less fees than expected by splitting up their position into multiple smaller positions. Let's imagine the scenario by using numbers, the scenario will be very simplified

and will ignore other types of fees that are incurred, we will only focus on the additive fee based on the utilization ratio.

Current state:

- usedPoints = 0
- maxPoints = 10_000

Scenario A: User creates a position for 10_000 tokens which causes the utilization ratio to go to 100%. The fee percentage to pay is based on the following formula:

```
return (utilizationPCT * maxFeeProjection) / DECIMAL_PRECISION;
```

As the `maxFeeProjection` is 5e16, then this will result in 5% of fees, thus 500 tokens of fees. This is the normal, happy path scenario.

Scenario B: User splits his position into 10 smaller positions, each with 1_000 tokens. The fee percentage will be the following for the 10 positions:

1. 1e17 * 5e16 / 1e18 = 5e15 (10% utilization ratio as the position is for 1_000 out of 10_000)
2. 2e17 * 5e16 / 1e18 = 1e16
3. 3e17 * 5e16 / 1e18 = 1.5e16
4. 4e17 * 5e16 / 1e18 = 2e16
5. 5e17 * 5e16 / 1e18 = 2.5e16
6. 6e17 * 5e16 / 1e18 = 3e16
7. 7e17 * 5e16 / 1e18 = 3.5e16
8. 8e17 * 5e16 / 1e18 = 4e16
9. 9e17 * 5e16 / 1e18 = 4.5e16
10. 1e18 * 5e16 / 1e18 = 5e16 (here, we are at 100% utilization ratio as we now have 10 positions with 1_000 tokens each)

The total fees will be `(5e15 * 1000 / 1e18) + (1e16 * 1000 / 1e18) + (1.5e16 * 1000 / 1e18) + (2e16 * 1000 / 1e18) + (2.5e16 * 1000 / 1e18) + (3e16 * 1000 / 1e18) + (3.5e16 * 1000 / 1e18) + (4e16 * 1000 / 1e18) + (4.5e16 * 1000 / 1e18) + (5e16 * 1000 / 1e18) = 275`, this is a 45% lower amount of fees compared to the 500 if the user just opens 1 position.

Issue is inherent in the protocol design of calculating fees based on a utilization ratio, fix is not trivial without introducing other issues to consider.

Either accept it as a design risk or remove the additive fees based on the utilization.

[L-09] Loss of precision in loan and redemption point regeneration slows the regeneration rate

The regeneration of loan and redemption points is calculated as follows:

```
File: CollateralControllerState.sol

370:         uint secondsElapsed = targetTimestamp - lastTimestamp;
371:
372:         (bool safeMul, uint _regeneratedPoints) = secondsElapsed.tryMul
  (regenerationRatePerMin);
373:         uint regeneratedPoints = (safeMul ? _regeneratedPoints : type
  (uint).max) / 60;
```

When `secondsElapsed * regenerationRatePerMin` is not a multiple of 60, the regenerated points will be rounded down, slowing the regeneration rate. For example, if `regenerationRatePerMin` is 3, and `secondsElapsed` is 12, the regenerated points will be 0.

It is recommended to replace the `regenerationRatePerMin` setting with a `regenerationRatePerSec` setting and remove the division by 60.

[L-10] Calls to Chainlink do not prevent griefing attacks

The `EthPriceFeed.fetchEthPrice()` function calls Chainlink's price aggregator to get the latest price of ETH. If the call to the price aggregator reverts Chainlink oracle is considered broken and its status is set accordingly, requiring the intervention of the guardian to restore it.

```
141:         try priceAggregator.latestRoundData() returns
142:         (
    (...)
156:         } catch {
157:             // If call to Chainlink aggregator reverts, return a zero
// response with success = false
158:             return chainlinkResponse;
159         }
```

There are two important things to note here:

- If the call to the target function reverts due to an out-of-gas exception, the exception is caught and the `catch` block is executed.
- Only 63/64 of the gas is forwarded to the target function.

This means that a grief attack would be possible if a malicious actor called the `fetchEthPrice()` function with a small amount of gas, provoking an out-of-gas exception and causing the Chainlink oracle to be considered broken.

It is important to note that for the main transaction not to revert due to an out-of-gas exception, the outstanding gas (1/64 of the gas forwarded) should be enough to execute the update of the Chainlink oracle status. Or expressed differently, the execution of `priceAggregator.latestRoundData()` needs to consume at least 64 times the gas consumed by the execution of `fetchEthPrice()` after the exception is caught. While this is currently not the case, given that the price aggregator contract acts as a proxy, this could change in the future and make the attack possible.

Consider implementing a solution similar to the one added in Liquity V2 to all the calls to the Chainlink oracle.

[L-11] Burning clawback rewards causes total debt to be greater than stable total supply

The `burnClawbackRewards` function in the `PositionController` contract burns the stable tokens minted to the contract on `claimEscrow`. However, the debt accounted for by the protocol has not been reduced.

This means that the total debt of the protocol (sum of debt in active and default pool for all collaterals) is not anymore equal to the total supply of stable

tokens. This results in a overestimation of the total debt in the system and thus, of the minimum collateral ratios.

A simplified example to illustrate the issue:

```
Initial state:
- Total supply: 100 USDx
- Total debt: 100 USDx
- Total collateral value: 200 USD
- TCR: 200%

After clawback rewards are claimed and collateral price decreases:
- Total supply: 60 USDx
- Total debt: 100 USDx
- Total collateral value: 120 USD
- TCR: 120%

While
   the collateral value backs all the stable tokens in circulation with a 200% ratio,
```

Consider removing this function and using in all cases the `distributeClawbackRewards` function to distribute the stables to the fee stakers.