Security Audit Report for Sumero Finance

Date: May 23, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2023.05.24
15:59:54 +08'00'

Contents

Report Manifest

Item	Description
Client	Sumero Finance
Target	Sumero Finance

Version History

Version	Date	Description
1.0	May 23, 2023	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The repository that has been audited includes sumero-contracts-0.1.4-pre-release [1].

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
Sumero Finance	`Version 1`	`681310bcba66c00fc3ae43de4d9ef40bc9a44567` (`v0.1.4-pre-release`)
	`Version 2`	`b887e47586f91cfe5d9495aaaf62ebe8acf5b7a6` (`v0.1.5-pre-release`)

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **contracts** folder contract only. Specifically, the files covered in this audit include:

- interfaces
- UMA/common
- UMA/financial-templates
- UMA/oracle
- uniswapV2
- AssetManager.sol
- ClayBonds.sol
- ClayDistributor.sol
- ClayStakingRewards.sol
- ClayToken.sol

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

[1] https://github.com/SumeroApp/sumero-contracts/releases/tag/v0.1.4-pre-release

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism

- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **three** potential issues. Besides, we have **eight** recommendations and **three** notes as follows:

- High Risk: 2
- Medium Risk: 0
- Low Risk: 1
- Recommendations: 8
- Notes: 3

ID	Severity	Description	Category	Status
1	High	Incorrect Calculation when Updating rewardRate	DeFi Security	Fixed
2	Low	Improper Check of maximumBondRewards	DeFi Security	Fixed
3	High	Lack of Check for Collateralization Rate of the first user in create()	DeFi Security	Acknowledged
4	-	Avoid Duplicated Assets in AssetManager	Recommendation	Confirmed
5	-	Avoid Incomplete Check of Whitelist Status	Recommendation	Confirmed
6	-	Code Optimization	Recommendation	Confirmed
7	-	Add Sanity Address Checks in Constructor	Recommendation	Fixed
8	-	Redundant Check in withdrawPassedRequest()	Recommendation	Confirmed
9	-	Add Sanity Checks for WithdrawLiveness	Recommendation	Confirmed
10	-	Add Sanity Checks for Transformed settlementPrice	Confirmed	Acknowledged
11	-	Redundant Check in createLiquidation()	Recommendation	Confirmed
12	-	Temporary System Parameters for Testing	Note	Fixed
13	-	Customized FinancialProductLibrary	Note	Confirmed
14	-	Potential Centralized Problem	Note	Confirmed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect Calculation when Updating rewardRate

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The privileged function `updateMaxReward()` is designed to update the `rewardRate` of the staking contract. In this function, the new `rewardRate` is calculated based on the remaining amount of rewards that need to be distributed within the remaining time.

However, the current implementation calculates the remaining rewards by subtracting the `rewardPerTokenStored` from `maxReward`, which is incorrect. This is because `rewardPerTokenStored` represents the amount of reward that can be claimed per share instead of the total distributed rewards.

```
168   function updateMaxReward(uint256 _maxReward) external onlyOwner notExpired {
169       rewardPerTokenStored = rewardPerToken();
170       require(
171           rewardPerTokenStored < _maxReward,
172           "ClayStakingRewards: INVALID_MAX_REWARD_AMOUNT"
173       );
174       lastUpdateTime = block.timestamp;
175       maxReward = _maxReward;
176       rewardRate =
177           (_maxReward - rewardPerTokenStored / 1e18) /
178           (periodFinish - block.timestamp);
179       emit RewardRateUpdated(rewardRate);
180    }
```

Listing 2.1: ClayStakingRewards.sol

Impact Rewards may be distributed more than expected.

Suggestion To calculate the rewards that have already been distributed, multiply the `rewardPerTokenStored` by `_totalSupply`.

2.1.2 Improper Check of maximumBondRewards

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `issue()` allows users to deposit their `CLAY`s for `zCLAY` bonds. The amount of `zCLAY` (i.e., `bondAmount`) minted to the user is the sum of initial deposited `CLAY`s and rewards.

According to the design, the total supply of minted `zCLAY` bonds should never exceed `maximumBondRewards` (i.e., less than or equal to). However, the check implemented in this function (line 144) makes sure that the total amount of `zCLAY` bonds is always less than `maximumBondRewards`, which is not consistent with the design.

```
113   function issue(uint256 _clayAmount) external returns (uint256 bondAmount) {
114       require(
115           _clayAmount > MIN_ISSUANCE_AMOUNT,
116           "ClayBonds: INSUFFICIENT_AMOUNT"
117       );
118       require(
119           block.timestamp >= depositStartDate,
120           "ClayBonds: DEPOSIT_NOT_YET_STARTED"
121       );
122       require(
123           block.timestamp < depositCloseDate,
124           "ClayBonds: DEPOSIT_CLOSED"
125       );
126
127       uint256 daysLeftToMaturationDate = getDaysLeftToMaturationDate();
128       uint256 rewardPercent = getRewardPercent(daysLeftToMaturationDate);
129       uint256 reward = getReward(_clayAmount, rewardPercent);
```

```
130
131         bondAmount = _clayAmount + reward;
132
133         bool success = clay.transferFrom(
134             msg.sender,
135             address(this),
136             _clayAmount
137         );
138         require(success, "ClayBonds: TRANSFER_FAILED");
139         _mint(msg.sender, bondAmount);
140
141         totalBondDeposits = totalBondDeposits + bondAmount;
142
143         require(
144             totalBondDeposits < maximumBondRewards,
145             "ClayBonds: MAX_BOND_REWARD_POOL_REACHED"
146         );
147
148         emit Issued(
149             msg.sender,
150             _clayAmount,
151             daysLeftToMaturationDate,
152             rewardPercent,
153             reward
154         );
155     }
```

Listing 2.2: ClayBonds.sol

Impact The total amount of `zCLAY` bonds can never reach `maximumBondRewards`.

Suggestion The check should be adjusted to "`totalBondDeposits` <= `maximumBondRewards`".

2.1.3 Lack of Check for Collateralization Rate of the first user in create()

Severity High

Status Acknowledged

Introduced by `Version 1`

Description The function `create()` allows the user to open a new position or increase an existing position with a valid collateralization ratio. However, the check for the collateralization ratio doesn't work for the first user who creates the position. In this case, the malicious user is able to mint an extremely large amount of `tokenCurrency` with only a small amount of collateral.

The first (malicious) position could be used to attack other valid positions (with valuable collateral) created by the follow-up users. Specifically, the malicious user can liquidate a valid position with a certain amount of `tokenCurrency` (via the function `createLiquidation()`) even if the position's collateralization ratio is eligible. It is worth noting that, even when a `disputer` succeeds to dispute, the cost of the malicious user (i.e., `disputerDisputeReward` and `sponsorDisputeReward`) is neglectable. Actually, the cost is much less than the profit, as `tokenCurrency` paid by the malicious user is minted with little cost.

```
483     function create(
```

```
484         FixedPoint.Unsigned memory collateralAmount,
485         FixedPoint.Unsigned memory numTokens
486     ) public onlyPreExpiration nonReentrant {
487         PositionData storage positionData = positions[msg.sender];
488 
489         // Either the new create ratio or the resultant position CR must be above the current GCR.
490         require(
491             (_checkCollateralization(
492                 positionData.collateral.add(collateralAmount),
493                 positionData.tokensOutstanding.add(numTokens)
494             ) || _checkCollateralization(collateralAmount, numTokens)),
495             "Insufficient collateral"
496         );
497 
498         require(
499             positionData.withdrawalRequestPassTimestamp == 0,
500             "Pending withdrawal"
501         );
502 
503         if (positionData.tokensOutstanding.isEqual(0)) {
504             require(
505                 numTokens.isGreaterThanOrEqual(minSponsorTokens),
506                 "Below minimum sponsor position"
507             );
508             emit NewSponsor(msg.sender);
509         }
510 
511         // Increase the position and global collateral balance by collateral amount.
512         _incrementCollateralBalances(positionData, collateralAmount);
513 
514         // Add the number of tokens created to the position's outstanding tokens.
515         positionData.tokensOutstanding = positionData.tokensOutstanding.add(
516             numTokens
517         );
518         totalTokensOutstanding = totalTokensOutstanding.add(numTokens);
519 
520         emit PositionCreated(
521             msg.sender,
522             collateralAmount.rawValue,
523             numTokens.rawValue
524         );
525 
526         // Transfer tokens into the contract from caller and mint corresponding synthetic tokens to
                the caller's address.
527         collateralCurrency.safeTransferFrom(
528             msg.sender,
529             address(this),
530             collateralAmount.rawValue
531         );
532         require(tokenCurrency.mint(msg.sender, numTokens.rawValue));
533     }
```

Listing 2.3: UMA/financial-templates/expiring-multiparty/PricelessPositionManager.sol

```
452     function withdrawLiquidation(uint256 liquidationId, address sponsor)
453         public
454         withdrawable(liquidationId, sponsor)
455         nonReentrant
456         returns (RewardsData memory)
457     {
458         LiquidationData storage liquidation = _getLiquidationData(
459             sponsor,
460             liquidationId
461         );
462 
463         // Settles the liquidation if necessary. This call will revert if the price has not
                resolved yet.
464         _settle(liquidationId, sponsor);
465 
466         // Calculate rewards as a function of the TRV.
467         FixedPoint.Unsigned memory tokenRedemptionValue = liquidation
468             .tokensOutstanding
469             .mul(liquidation.settlementPrice);
470         FixedPoint.Unsigned
471             memory disputerDisputeReward = disputerDisputeRewardPercentage.mul(
472                 tokenRedemptionValue
473             );
474         FixedPoint.Unsigned
475             memory sponsorDisputeReward = sponsorDisputeRewardPercentage.mul(
476                 tokenRedemptionValue
477             );
478         FixedPoint.Unsigned memory disputeBondAmount = liquidation
479             .lockedCollateral
480             .mul(disputeBondPercentage);
481 
482         // There are three main outcome states: either the dispute succeeded, failed or was not
                updated.
483         // Based on the state, different parties of a liquidation receive different amounts.
484         // After assigning rewards based on the liquidation status, decrease the total collateral
                held in this contract
485         // by the amount to pay each party. The actual amounts withdrawn might differ if
                _removeCollateral causes
486         // precision loss.
487         RewardsData memory rewards;
488         if (liquidation.state == Status.DisputeSucceeded) {
489             // If the dispute is successful then all three users should receive rewards:
490 
491             // Pay DISPUTER: disputer reward + dispute bond + returned ooReward
492             rewards.payToDisputer = disputerDisputeReward
493                 .add(disputeBondAmount)
494                 .add(ooReward);
495 
496             // Pay SPONSOR: remaining collateral (collateral - TRV) + sponsor reward
497             rewards.payToSponsor = liquidation
498                 .lockedCollateral
499                 .sub(tokenRedemptionValue)
```

```
500                 .add(sponsorDisputeReward);
501
502             // Pay LIQUIDATOR: TRV - dispute reward - sponsor reward
503             // If TRV > Collateral, then subtract rewards from collateral
504             // NOTE: `payToLiquidator` should never be below zero since we enforce that
505             // (sponsorDisputePct+disputerDisputePct) <= 1 in the constructor when these params are
                    set.
506             rewards.payToLiquidator = tokenRedemptionValue
507                 .sub(sponsorDisputeReward)
508                 .sub(disputerDisputeReward);
509
510             // Transfer rewards and debit collateral
511             liquidationCollateral = liquidationCollateral.sub(
512                 rewards.payToLiquidator
513             );
514             liquidationCollateral = liquidationCollateral.sub(
515                 rewards.payToSponsor
516             );
517             liquidationCollateral = liquidationCollateral.sub(
518                 rewards.payToDisputer
519             );
520
521             collateralCurrency.safeTransfer(
522                 liquidation.disputer,
523                 rewards.payToDisputer.rawValue
524             );
525             collateralCurrency.safeTransfer(
526                 liquidation.liquidator,
527                 rewards.payToLiquidator.rawValue
528             );
529             collateralCurrency.safeTransfer(
530                 liquidation.sponsor,
531                 rewards.payToSponsor.rawValue
532             );
533         } else if (liquidation.state == Status.DisputeFailed) {
534             // In the case of a failed dispute only the liquidator can withdraw.
535
536             // Pay LIQUIDATOR: collateral + dispute bond + returned ooReward
537             rewards.payToLiquidator = liquidation
538                 .lockedCollateral
539                 .add(disputeBondAmount)
540                 .add(ooReward);
541
542             // Transfer rewards and debit collateral
543             liquidationCollateral = liquidationCollateral.sub(
544                 rewards.payToLiquidator
545             );
546
547             collateralCurrency.safeTransfer(
548                 liquidation.liquidator,
549                 rewards.payToLiquidator.rawValue
550             );
551         } else if (liquidation.state == Status.NotDisputed) {
```

```
552          // If the state is pre-dispute but time has passed liveness then there was no dispute.
                 We represent this
553          // state as a dispute failed and the liquidator can withdraw.
554
555          // Pay LIQUIDATOR: collateral + returned ooReward
556          rewards.payToLiquidator = liquidation.lockedCollateral.add(
557              ooReward
558          );
```

Listing 2.4: UMA/financial-templates/expiring-multiparty/Liquidatable.sol

Impact Other users' collateral may be maliciously liquidated by the first user.

Suggestion Add an additional check to make sure the collateralization ratio is valid when creating a new position.

Feedback from the Project `Sumero` would be deploying the `EMP` contract and then minting an initial amount of `Synth` (first position). Thus, always maintaining the correct global collateralization ratio (`GCR`). So, we believe this is not a valid issue with regards to the way we are using `EMP`.

2.2 Additional Recommendation

2.2.1 Avoid Duplicated Assets in AssetManager

Status Confirmed

Introduced by `Version 1`

Description The `owner` of the `AssetManager` has the ability to add new assets (i.e., `Emp`, `SwapPair`, and `StakingReward`) into the system. However, there is no check to ensure the newly added assets are not duplicated. Meanwhile, the global variables `totalEmpAssets`/`totalSwapPairAssets`/`totalStakingRewardAssets` will always increase, which is incorrect.

```
34   function addEmp(address _asset) external onlyOwner {
35       require(_asset != address(0), "AssetManager: ZERO_ADDRESS");
36       totalEmpAssets = totalEmpAssets + 1;
37       idToVerifiedEmps[totalEmpAssets] = Asset(_asset, Status.Open);
38       emit Added(Type.Emp, _asset, totalEmpAssets);
39    }
```

Listing 2.5: AssetManager.sol

```
73   function addSwapPair(address _asset) external onlyOwner {
74       require(_asset != address(0), "AssetManager: ZERO_ADDRESS");
75       totalSwapPairAssets = totalSwapPairAssets + 1;
76       idToVerifiedSwapPairs[totalSwapPairAssets] = Asset(_asset, Status.Open);
77       emit Added(Type.SwapPair, _asset, totalSwapPairAssets);
78    }
```

Listing 2.6: AssetManager.sol

```
112  function addStakingReward(address _asset) external onlyOwner {
113      require(_asset != address(0), "AssetManager: ZERO_ADDRESS");
```

```
114         totalStakingRewardAssets = totalStakingRewardAssets + 1;
115         idToVerifiedStakingRewards[totalStakingRewardAssets] = Asset(
116             _asset,
117             Status.Open
118         );
119         emit Added(Type.StakingReward, _asset, totalStakingRewardAssets);
120     }
```

Listing 2.7: AssetManager.sol

Suggestion I Add the check to make sure there exists no duplicated assets.

Feedback from the Project Noted. This is a design choice by us. `AssetManager` will only be used by `Sumero` team, to display assets on `UI`. We will make sure not to add duplicates. We will not have more than 10-15 assets in each of the global arrays, so length of the global arrays isn't a concern for us.

2.2.2 Avoid Incomplete Check of Whitelist Status

Status Confirmed

Introduced by `Version 1`

Description The function `removeFromWhitelist()` allows the `owner` to remove an address from the whitelist. There is a check in the function to ensure the existence of `elementToRemove` in the whitelist (line 59), but the `Status None` is also included, which is not reasonable.

```
53  function removeFromWhitelist(address elementToRemove)
54         external
55         override
56         nonReentrant
57         onlyOwner
58     {
59         if (whitelist[elementToRemove] != Status.Out) {
60             whitelist[elementToRemove] = Status.Out;
61             emit RemovedFromWhitelist(elementToRemove);
62         }
63     }
```

Listing 2.8: UMA/common/implementation/AddressWhitelist.sol

Suggestion I Make sure the `Status` of the `elementToRemove` is `In`.

Feedback from the Project Agreed. Valid point. We have forked `UMA` contracts, so we would want to avoid any changes unless absolutely necessary.

2.2.3 Code Optimization

Status Confirmed

Introduced by `Version 1`

Description In the contract `PricelessPositionManager`, the user can repay minted `tokenCurrency` back to the contract to increase the collateralization ratio of the position. The contract will then burn the received `tokenCurrency`. These two steps can be refactored to one step by invoking the function `burnFrom()` in the

tokenCurrency contract (i.e., ExpandedERC20.sol), which enables the Burner to burn tokens from the user directly.

Similar problems also exist in function redeem(), settleExpired(), and createLiquidation().

```
542    function repay(FixedPoint.Unsigned memory numTokens)
543        public
544        onlyPreExpiration
545        noPendingWithdrawal(msg.sender)
546        nonReentrant
547    {
548        PositionData storage positionData = _getPositionData(msg.sender);
549        require(numTokens.isLessThanOrEqual(positionData.tokensOutstanding));
550
551        // Decrease the sponsors position tokens size. Ensure it is above the min sponsor size.
552        FixedPoint.Unsigned memory newTokenCount = positionData
553            .tokensOutstanding
554            .sub(numTokens);
555        require(newTokenCount.isGreaterThanOrEqual(minSponsorTokens));
556        positionData.tokensOutstanding = newTokenCount;
557
558        // Update the totalTokensOutstanding after redemption.
559        totalTokensOutstanding = totalTokensOutstanding.sub(numTokens);
560
561        emit Repay(msg.sender, numTokens.rawValue, newTokenCount.rawValue);
562
563        // Transfer the tokens back from the sponsor and burn them.
564        tokenCurrency.safeTransferFrom(
565            msg.sender,
566            address(this),
567            numTokens.rawValue
568        );
569        tokenCurrency.burn(numTokens.rawValue);
570    }
```

Listing 2.9: UMA/financial-templates/expiring-multiparty/PricelessPositionManager.sol

```
581    function redeem(FixedPoint.Unsigned memory numTokens)
582        public
583        noPendingWithdrawal(msg.sender)
584        nonReentrant
585        returns (FixedPoint.Unsigned memory amountWithdrawn)
586    {
587        PositionData storage positionData = _getPositionData(msg.sender);
588        require(!numTokens.isGreaterThan(positionData.tokensOutstanding));
589
590        FixedPoint.Unsigned memory fractionRedeemed = numTokens.div(
591            positionData.tokensOutstanding
592        );
593        FixedPoint.Unsigned memory collateralRedeemed = fractionRedeemed.mul(
594            positionData.collateral
595        );
596
597        // If redemption returns all tokens the sponsor has then we can delete their position. Else
```

```
            , downsize.
598         if (positionData.tokensOutstanding.isEqual(numTokens)) {
599             amountWithdrawn = _deleteSponsorPosition(msg.sender);
600         } else {
601             // Decrement the sponsor's collateral and global collateral amounts.
602             amountWithdrawn = _decrementCollateralBalances(
603                 positionData,
604                 collateralRedeemed
605             );
606
607             // Decrease the sponsors position tokens size. Ensure it is above the min sponsor size.
608             FixedPoint.Unsigned memory newTokenCount = positionData
609                 .tokensOutstanding
610                 .sub(numTokens);
611             require(
612                 newTokenCount.isGreaterThanOrEqual(minSponsorTokens),
613                 "Below minimum sponsor position"
614             );
615             positionData.tokensOutstanding = newTokenCount;
616
617             // Update the totalTokensOutstanding after redemption.
618             totalTokensOutstanding = totalTokensOutstanding.sub(numTokens);
619         }
620
621         emit Redeem(msg.sender, amountWithdrawn.rawValue, numTokens.rawValue);
622
623         // Transfer collateral from contract to caller and burn callers synthetic tokens.
624         collateralCurrency.safeTransfer(msg.sender, amountWithdrawn.rawValue);
625         tokenCurrency.safeTransferFrom(
626             msg.sender,
627             address(this),
628             numTokens.rawValue
629         );
630         tokenCurrency.burn(numTokens.rawValue);
631     }
```

Listing 2.10: UMA/financial-templates/expiring-multiparty/PricelessPositionManager.sol

```
642     function settleExpired()
643         external
644         onlyPostExpiration
645         nonReentrant
646         returns (FixedPoint.Unsigned memory amountWithdrawn)
647     {
648         // If the contract state is open and onlyPostExpiration passed then `expire()` has not yet
                been called.
649         require(contractState != ContractState.Open, "Unexpired position");
650
651         // Get the current settlement price and store it. If it is not resolved will revert.
652         if (contractState != ContractState.ExpiredPriceReceived) {
653             expiryPrice = _getOraclePrice(expirationTimestamp);
654             contractState = ContractState.ExpiredPriceReceived;
655         }
```

```
656
657         // Get caller's tokens balance and calculate amount of underlying entitled to them.
658         FixedPoint.Unsigned memory tokensToRedeem = FixedPoint.Unsigned(
659             tokenCurrency.balanceOf(msg.sender)
660         );
661
662         FixedPoint.Unsigned memory totalRedeemableCollateral = tokensToRedeem
663             .mul(expiryPrice);
664
665         // If the caller is a sponsor with outstanding collateral they are also entitled to their
                excess collateral after their debt.
666         PositionData storage positionData = positions[msg.sender];
667         if (positionData.collateral.isGreaterThan(0)) {
668             // Calculate the underlying entitled to a token sponsor. This is collateral - debt in
                    underlying.
669             FixedPoint.Unsigned memory tokenDebtValueInCollateral = positionData
670                 .tokensOutstanding
671                 .mul(expiryPrice);
672             FixedPoint.Unsigned memory positionCollateral = positionData
673                 .collateral;
674
675             // If the debt is greater than the remaining collateral, they cannot redeem anything.
676             FixedPoint.Unsigned
677                 memory positionRedeemableCollateral = tokenDebtValueInCollateral
678                     .isLessThan(positionCollateral)
679                     ? positionCollateral.sub(tokenDebtValueInCollateral)
680                     : FixedPoint.Unsigned(0);
681
682             // Add the number of redeemable tokens for the sponsor to their total redeemable
                    collateral.
683             totalRedeemableCollateral = totalRedeemableCollateral.add(
684                 positionRedeemableCollateral
685             );
686
687             // Reset the position state as all the value has been removed after settlement.
688             delete positions[msg.sender];
689             emit EndedSponsorPosition(msg.sender);
690         }
691
692         // Take the min of the remaining collateral and the collateral "owed". If the contract is
                undercapitalized,
693         // the caller will get as much collateral as the contract can pay out.
694         FixedPoint.Unsigned memory payout = FixedPoint.min(
695             totalPositionCollateral,
696             totalRedeemableCollateral
697         );
698
699         // Decrement total contract collateral and outstanding debt.
700         totalPositionCollateral = totalPositionCollateral.sub(payout);
701         amountWithdrawn = payout;
702         totalTokensOutstanding = totalTokensOutstanding.sub(tokensToRedeem);
703
704         emit SettleExpiredPosition(
```

```
705             msg.sender,
706             amountWithdrawn.rawValue,
707             tokensToRedeem.rawValue
708         );
709
710         // Transfer tokens & collateral and burn the redeemed tokens.
711         collateralCurrency.safeTransfer(msg.sender, amountWithdrawn.rawValue);
712         tokenCurrency.safeTransferFrom(
713             msg.sender,
714             address(this),
715             tokensToRedeem.rawValue
716         );
717         tokenCurrency.burn(tokensToRedeem.rawValue);
718     }
```

Listing 2.11: UMA/financial-templates/expiring-multiparty/PricelessPositionManager.sol

```
226     function createLiquidation(
227         address sponsor,
228         FixedPoint.Unsigned calldata minCollateralPerToken,
229         FixedPoint.Unsigned calldata maxCollateralPerToken,
230         FixedPoint.Unsigned calldata maxTokensToLiquidate,
231         uint256 deadline
232     )
233         external
234         onlyPreExpiration
235         nonReentrant
236         returns (
237             uint256 liquidationId,
238             FixedPoint.Unsigned memory tokensLiquidated
239         )
240     {
241         // Check that this transaction was mined pre-deadline.
242         require(block.timestamp <= deadline, "Mined after deadline");
243
244         // Retrieve Position data for sponsor
245         PositionData storage positionToLiquidate = _getPositionData(sponsor);
246
247         tokensLiquidated = FixedPoint.min(
248             maxTokensToLiquidate,
249             positionToLiquidate.tokensOutstanding
250         );
251         require(tokensLiquidated.isGreaterThan(0));
252
253         // Starting values for the Position being liquidated. If withdrawal request amount is >
                position's collateral,
254         // then set this to 0, otherwise set it to (startCollateral - withdrawal request amount).
255         FixedPoint.Unsigned memory startCollateral = positionToLiquidate
256             .collateral;
257         FixedPoint.Unsigned memory startCollateralNetOfWithdrawal = FixedPoint
258             .fromUnscaledUint(0);
259         if (
260             positionToLiquidate.withdrawalRequestAmount.isLessThanOrEqual(
```

```
261                 startCollateral
262             )
263         ) {
264             startCollateralNetOfWithdrawal = startCollateral.sub(
265                 positionToLiquidate.withdrawalRequestAmount
266             );
267         }
268
269         // Scoping to get rid of a stack too deep error.
270         {
271             FixedPoint.Unsigned memory startTokens = positionToLiquidate
272                 .tokensOutstanding;
273
274             // The Position's collateralization ratio must be between [minCollateralPerToken,
                    maxCollateralPerToken].
275             // maxCollateralPerToken >= startCollateralNetOfWithdrawal / startTokens.
276             require(
277                 maxCollateralPerToken.mul(startTokens).isGreaterThanOrEqual(
278                     startCollateralNetOfWithdrawal
279                 ),
280                 "CR is more than max liq. price"
281             );
282             // minCollateralPerToken <= startCollateralNetOfWithdrawal / startTokens.
283             require(
284                 minCollateralPerToken.mul(startTokens).isLessThanOrEqual(
285                     startCollateralNetOfWithdrawal
286                 ),
287                 "CR is less than min liq. price"
288             );
289         }
290
291         // These will be populated within the scope below.
292         FixedPoint.Unsigned memory lockedCollateral;
293         FixedPoint.Unsigned memory lockedCollateralAfterWithdrawals;
294
295         // Scoping to get rid of a stack too deep error.
296         {
297             FixedPoint.Unsigned memory ratio = tokensLiquidated.div(
298                 positionToLiquidate.tokensOutstanding
299             );
300
301             // The actual amount of collateral that gets moved to the liquidation.
302             lockedCollateral = startCollateral.mul(ratio);
303
304             // For purposes of disputes, it's actually this lockedCollateralAfterWithdrawals value
                    that's used.
305             lockedCollateralAfterWithdrawals = startCollateralNetOfWithdrawal
306                 .mul(ratio);
307
308             // Part of the withdrawal request is also removed. Ideally:
309             // lockedCollateralAfterWithdrawals + withdrawalAmountToRemove = lockedCollateral.
310             FixedPoint.Unsigned
311                 memory withdrawalAmountToRemove = positionToLiquidate
```

```
312                 .withdrawalRequestAmount
313                 .mul(ratio);
314
315         _reduceSponsorPosition(
316             sponsor,
317             tokensLiquidated,
318             lockedCollateral,
319             withdrawalAmountToRemove
320         );
321     }
322
323     // Add to the global liquidation collateral count.
324     liquidationCollateral = liquidationCollateral.add(lockedCollateral).add(
325             ooReward
326         );
327
328     // Construct liquidation object.
329     // Note: All dispute-related values are zeroed out until a dispute occurs. liquidationId is
            the index of the new
330     // LiquidationData that is pushed into the array, which is equal to the current length of
            the array pre-push.
331     liquidationId = liquidations[sponsor].length;
332     liquidations[sponsor].push(
333         LiquidationData({
334             sponsor: sponsor,
335             liquidator: msg.sender,
336             state: Status.NotDisputed,
337             liquidationTime: block.timestamp,
338             tokensOutstanding: tokensLiquidated,
339             lockedCollateral: lockedCollateral,
340             lockedCollateralAfterWithdrawals: lockedCollateralAfterWithdrawals,
341             disputer: address(0),
342             settlementPrice: FixedPoint.fromUnscaledUint(0)
343         })
344     );
345
346     // If this liquidation is a subsequent liquidation on the position, and the liquidation
            size is larger than
347     // some "griefing threshold", then re-set the liveness. This enables a liquidation against
            a withdraw request to be
348     // "dragged out" if the position is very large and liquidators need time to gather funds.
            The griefing threshold
349     // is enforced so that liquidations for trivially small # of tokens cannot drag out an
            honest sponsor's slow withdrawal.
350
351     // We arbitrarily set the "griefing threshold" to `minSponsorTokens` because it is the only
            parameter
352     // denominated in token currency units and we can avoid adding another parameter.
353     FixedPoint.Unsigned memory griefingThreshold = minSponsorTokens;
354     if (
355         positionToLiquidate.withdrawalRequestPassTimestamp > 0 && // The position is undergoing
                a slow withdrawal.
356         positionToLiquidate.withdrawalRequestPassTimestamp >
```

```
357             block.timestamp && // The slow withdrawal has not yet expired.
358             tokensLiquidated.isGreaterThanOrEqual(griefingThreshold) // The liquidated token count
                    is above a "griefing threshold".
359         ) {
360             positionToLiquidate.withdrawalRequestPassTimestamp = (
361                 block.timestamp
362             ).add(withdrawalLiveness);
363         }
364
365         emit LiquidationCreated(
366             sponsor,
367             msg.sender,
368             liquidationId,
369             tokensLiquidated.rawValue,
370             lockedCollateral.rawValue,
371             lockedCollateralAfterWithdrawals.rawValue,
372             block.timestamp
373         );
374
375         // Destroy tokens
376         tokenCurrency.safeTransferFrom(
377             msg.sender,
378             address(this),
379             tokensLiquidated.rawValue
380         );
381         tokenCurrency.burn(tokensLiquidated.rawValue);
382
383         // Pull ooReward from liquidator.
384         collateralCurrency.safeTransferFrom(
385             msg.sender,
386             address(this),
387             ooReward.rawValue
388         );
389     }
```

Listing 2.12: UMA/financial-templates/expiring-multiparty/Liquidatable.sol

```
91  function burnFrom(address recipient, uint256 value)
92      external
93      override
94      onlyRoleHolder(uint256(Roles.Burner))
95      returns (bool)
96  {
97      _burn(recipient, value);
98      return true;
99  }
```

Listing 2.13: UMA/common/implementation/ExpandedERC20.sol

Suggestion I Implement the function `burnFrom()` instead of `safetransferFrom()` and `burn()`.

2.2.4 Add Sanity Address Checks in Constructor

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `constructor` of the contract `ClayStakingRewards`, there is no check to ensure that the addresses of `stakingToken` and `clayToken` are not zero.

Similar problems also exist in the contract `ClayDistributor`.

```
53    constructor(
54        address _stakedToken,
55        address _clayToken,
56        uint256 _periodFinish,
57        uint256 _maxReward
58    ) {
59        stakingToken = IERC20(_stakedToken);
60        clayToken = IClayToken(_clayToken);
61        periodFinish = _periodFinish;
62        maxReward = _maxReward;
63        rewardRate = _maxReward / (_periodFinish - block.timestamp);
64    }
```

Listing 2.14: ClayStakingRewards.sol

```
16    constructor(address token_, bytes32 merkleRoot_, uint256 dropAmount_) {
17        token = token_;
18        merkleRoot = merkleRoot_;
19        dropAmount = dropAmount_;
20    }
```

Listing 2.15: ClayDistributor.sol

Suggestion I Add the check to ensure the addresses added in the `constructor` are not zero.

2.2.5 Redundant Check in withdrawPassedRequest()

Status Confirmed

Introduced by `Version 1`

Description The function `requestWithdrawal()` allows the user to request for a withdrawal to withdraw a certain amount of collateral from the position. The request has to be pending for a period of `withdrawlLiveness`, after which the user can invoke the function `withdrawPassedRequest()` to withdraw the requested collateral.

In the function `requestWithdrawal()`, there is a check to ensure that the requested withdrawal amount will always be less or equal than the total collateral in the position (lines 397 - 398). However, this same check also exists in the function `withdrawPassedRequest()`(lines 434 - 435). Since the collateral will never decrease during the pending period, the check in the function `withdrawPassedRequest()` is redundant.

```
389  function requestWithdrawal(FixedPoint.Unsigned memory collateralAmount)
390        public
391        onlyPreExpiration
392        noPendingWithdrawal(msg.sender)
```

```
393         nonReentrant
394     {
395         PositionData storage positionData = _getPositionData(msg.sender);
396         require(
397             collateralAmount.isGreaterThan(0) &&
398                 collateralAmount.isLessThanOrEqual(positionData.collateral)
399         );
400
401         // Make sure the proposed expiration of this request is not post-expiry.
402         uint256 requestPassTime = (block.timestamp).add(withdrawalLiveness);
403         require(requestPassTime < expirationTimestamp);
404
405         // Update the position object for the user.
406         positionData.withdrawalRequestPassTimestamp = requestPassTime;
407         positionData.withdrawalRequestAmount = collateralAmount;
408
409         emit RequestWithdrawal(msg.sender, collateralAmount.rawValue);
410     }
```

Listing 2.16: UMA/financial-templates/expiring-multiparty/PricelessPositionManager.sol

```
419     function withdrawPassedRequest()
420         external
421         onlyPreExpiration
422         nonReentrant
423         returns (FixedPoint.Unsigned memory amountWithdrawn)
424     {
425         PositionData storage positionData = _getPositionData(msg.sender);
426         require(
427             positionData.withdrawalRequestPassTimestamp != 0 &&
428                 positionData.withdrawalRequestPassTimestamp <= block.timestamp
429         );
430
431         // If withdrawal request amount is > position collateral, then withdraw the full collateral
                amount.
432         FixedPoint.Unsigned memory amountToWithdraw;
433         if (
434             positionData.withdrawalRequestAmount.isGreaterThan(
435                 positionData.collateral
436             )
437         ) {
438             amountToWithdraw = positionData.collateral;
439         } else {
440             amountToWithdraw = positionData.withdrawalRequestAmount;
441         }
442
443         // Decrement the sponsor's collateral and global collateral amounts.
444         amountWithdrawn = _decrementCollateralBalances(
445             positionData,
446             amountToWithdraw
447         );
448
449         // Reset withdrawal request by setting withdrawal amount and withdrawal timestamp to 0.
```

```
450         _resetWithdrawalRequest(positionData);
451
452         // Transfer approved withdrawal amount from the contract to the caller.
453         collateralCurrency.safeTransfer(msg.sender, amountWithdrawn.rawValue);
454
455         emit RequestWithdrawalExecuted(msg.sender, amountWithdrawn.rawValue);
456     }
```

Listing 2.17: UMA/financial-templates/expiring-multiparty/PricelessPositionManager.sol

Suggestion I Remove the redundant check in the function `withdrawPassedRequest()`.

2.2.6 Add Sanity Checks for WithdrawLiveness

Status Confirmed

Introduced by `Version 1`

Description In the contract `PricelessPostionManager`, there is a global parameter named `withdrawalLiveness` which is assigned in the constructor by the creator. This parameter specifies the time period for a `sponsor` to transfer position or withdraw all collateral. Specifically, the contract design requires the `sponsor` to first request for a withdrawal or transfer, and then wait for a certain period of time (i.e., `withdrawlLiveness`) before being able to transfer or withdraw.

Note that the contract requires the withdrawal or transfer time to be less than the expiration time of the contract (i.e., `expirationTimestamp`), so in the constructor, a check should be added to ensure that from the timestamp when the contract is deployed, the contract will not expire after a period of `withdrawalLiveness`.

```
209     constructor(
210         uint256 _expirationTimestamp,
211         uint256 _withdrawalLiveness,
212         address _collateralAddress,
213         address _tokenAddress,
214         address _finderAddress,
215         bytes32 _priceIdentifier,
216         FixedPoint.Unsigned memory _minSponsorTokens,
217         FixedPoint.Unsigned memory _ooReward,
218         address _financialProductLibraryAddress,
219         bytes memory _ancillaryData,
220         address _owner
221     ) nonReentrant() {
222         finder = FinderInterface(_finderAddress);
223
224         require(_expirationTimestamp > block.timestamp);
225         require(
226             _getIdentifierWhitelist().isIdentifierSupported(_priceIdentifier)
227         );
228
229         expirationTimestamp = _expirationTimestamp;
230         withdrawalLiveness = _withdrawalLiveness;
231         tokenCurrency = ExpandedIERC20(_tokenAddress);
232         collateralCurrency = IERC20(_collateralAddress);
233         minSponsorTokens = _minSponsorTokens;
234         ooReward = _ooReward;
```

```
235         priceIdentifier = _priceIdentifier;
236         ancillaryData = _ancillaryData;
237         owner = _owner;
238
239         // Initialize the financialProductLibrary at the provided address.
240         financialProductLibrary = FinancialProductLibrary(
241             _financialProductLibraryAddress
242         );
243     }
```

Listing 2.18: UMA/financial-templates/expiring-multiparty/PricelessPositionManager.sol

Suggestion I Add a check in the constructor to ensure "(`block.timestamp`).add(`withdrawalLiveness`) < `expirationTimestamp`"

2.2.7 Add Sanity Checks for Transformed settlementPrice

Status Confirmed

Introduced by `Version 1`

Description The internal function `_settle()` will settle a liquidation that is in the `Disputed` state. The required collateral is calculated as the value of the underlying collateral multiplied by the required collateral ratio. If the position has more collateral than the contract requires, then the liquidation is invalid.

In the current implementation, the required collateral ratio is the same as `collateralRequirement`. However, if the `financialProductLibrary` is deployed customly by the creator, the returned required collateral ratio may be different. In this case, there should be a check to make sure the ratio is always larger than one.

```
622     function _settle(uint256 liquidationId, address sponsor) internal {
623         LiquidationData storage liquidation = _getLiquidationData(
624             sponsor,
625             liquidationId
626         );
627
628         // Settlement only happens when state == Disputed and will only happen once per liquidation
                .
629         // If this liquidation is not ready to be settled, this method should return immediately.
630         if (liquidation.state != Status.Disputed) {
631             return;
632         }
633
634         // Get the returned price from the oracle. If this has not yet resolved will revert.
635         liquidation.settlementPrice = _getOraclePrice(
636             liquidation.liquidationTime
637         );
638
639         // Find the value of the tokens in the underlying collateral.
640         FixedPoint.Unsigned memory tokenRedemptionValue = liquidation
641             .tokensOutstanding
642             .mul(liquidation.settlementPrice);
643
```

```
644         // The required collateral is the value of the tokens in underlying * required collateral
                ratio. The Transform
645         // Collateral requirement method applies a from the financial Product library to change the
                scaled the collateral
646         // requirement based on the settlement price. If no library was specified when deploying
                the emp then this makes no change.
647         FixedPoint.Unsigned memory requiredCollateral = tokenRedemptionValue
648             .mul(_transformCollateralRequirement(liquidation.settlementPrice));
649
650         // If the position has more than the required collateral it is solvent and the dispute is
                valid(liquidation is invalid)
651         // Note that this check uses the lockedCollateralAfterWithdrawals not the lockedCollateral
                as this considers withdrawals.
652         bool disputeSucceeded = liquidation
653             .lockedCollateralAfterWithdrawals
654             .isGreaterThanOrEqual(requiredCollateral);
655
656         liquidation.state = disputeSucceeded
657             ? Status.DisputeSucceeded
658             : Status.DisputeFailed;
659
660         emit DisputeSettled(
661             msg.sender,
662             sponsor,
663             liquidation.liquidator,
664             liquidation.disputer,
665             liquidationId,
666             disputeSucceeded
667         );
668     }
```

Listing 2.19: UMA/financial-templates/expiring-multiparty/Liquidatable.sol

```
729     function _transformCollateralRequirement(FixedPoint.Unsigned memory price)
730         internal
731         view
732         returns (FixedPoint.Unsigned memory)
733     {
734         if (!address(financialProductLibrary).isContract())
735             return collateralRequirement;
736         try
737             financialProductLibrary.transformCollateralRequirement(
738                 price,
739                 collateralRequirement
740             )
741         returns (FixedPoint.Unsigned memory transformedCollateralRequirement) {
742             return transformedCollateralRequirement;
743         } catch {
744             return collateralRequirement;
745         }
746     }
```

Listing 2.20: UMA/financial-templates/expiring-multiparty/Liquidatable.sol

Suggestion I Add a check in the function `_transformCollateralRequirement()` to make sure `transformedCollateral` is larger than one.

2.2.8 Redundant Check in createLiquidation()

Status Confirmed

Introduced by `Version 1`

Description In function `createLiquidation()`, there is a check on whether the position has a pending withdrawal request. However, when the `withdrawalRequestPassTimestamp` is larger than the current timestamp, it must be larger than 0. Thus, the check that `withdrawalRequestPassTimestamp` should be larger than zero is redundant.

```
226    function createLiquidation(
227        address sponsor,
228        FixedPoint.Unsigned calldata minCollateralPerToken,
229        FixedPoint.Unsigned calldata maxCollateralPerToken,
230        FixedPoint.Unsigned calldata maxTokensToLiquidate,
231        uint256 deadline
232    )
233        external
234        onlyPreExpiration
235        nonReentrant
236        returns (
237            uint256 liquidationId,
238            FixedPoint.Unsigned memory tokensLiquidated
239        )
240    {
241        // Check that this transaction was mined pre-deadline.
242        require(block.timestamp <= deadline, "Mined after deadline");
243
244        // Retrieve Position data for sponsor
245        PositionData storage positionToLiquidate = _getPositionData(sponsor);
246
247        tokensLiquidated = FixedPoint.min(
248            maxTokensToLiquidate,
249            positionToLiquidate.tokensOutstanding
250        );
251        require(tokensLiquidated.isGreaterThan(0));
252
253        // Starting values for the Position being liquidated. If withdrawal request amount is >
                position's collateral,
254        // then set this to 0, otherwise set it to (startCollateral - withdrawal request amount).
255        FixedPoint.Unsigned memory startCollateral = positionToLiquidate
256            .collateral;
257        FixedPoint.Unsigned memory startCollateralNetOfWithdrawal = FixedPoint
258            .fromUnscaledUint(0);
259        if (
260            positionToLiquidate.withdrawalRequestAmount.isLessThanOrEqual(
261                startCollateral
262            )
263        ) {
```

```
264             startCollateralNetOfWithdrawal = startCollateral.sub(
265                 positionToLiquidate.withdrawalRequestAmount
266             );
267         }
268
269         // Scoping to get rid of a stack too deep error.
270         {
271             FixedPoint.Unsigned memory startTokens = positionToLiquidate
272                 .tokensOutstanding;
273
274             // The Position's collateralization ratio must be between [minCollateralPerToken,
                    maxCollateralPerToken].
275             // maxCollateralPerToken >= startCollateralNetOfWithdrawal / startTokens.
276             require(
277                 maxCollateralPerToken.mul(startTokens).isGreaterThanOrEqual(
278                     startCollateralNetOfWithdrawal
279                 ),
280                 "CR is more than max liq. price"
281             );
282             // minCollateralPerToken <= startCollateralNetOfWithdrawal / startTokens.
283             require(
284                 minCollateralPerToken.mul(startTokens).isLessThanOrEqual(
285                     startCollateralNetOfWithdrawal
286                 ),
287                 "CR is less than min liq. price"
288             );
289         }
290
291         // These will be populated within the scope below.
292         FixedPoint.Unsigned memory lockedCollateral;
293         FixedPoint.Unsigned memory lockedCollateralAfterWithdrawals;
294
295         // Scoping to get rid of a stack too deep error.
296         {
297             FixedPoint.Unsigned memory ratio = tokensLiquidated.div(
298                 positionToLiquidate.tokensOutstanding
299             );
300
301             // The actual amount of collateral that gets moved to the liquidation.
302             lockedCollateral = startCollateral.mul(ratio);
303
304             // For purposes of disputes, it's actually this lockedCollateralAfterWithdrawals value
                    that's used.
305             lockedCollateralAfterWithdrawals = startCollateralNetOfWithdrawal
306                 .mul(ratio);
307
308             // Part of the withdrawal request is also removed. Ideally:
309             // lockedCollateralAfterWithdrawals + withdrawalAmountToRemove = lockedCollateral.
310             FixedPoint.Unsigned
311                 memory withdrawalAmountToRemove = positionToLiquidate
312                     .withdrawalRequestAmount
313                     .mul(ratio);
314
```

```
315             _reduceSponsorPosition(
316                 sponsor,
317                 tokensLiquidated,
318                 lockedCollateral,
319                 withdrawalAmountToRemove
320             );
321         }
322 
323         // Add to the global liquidation collateral count.
324         liquidationCollateral = liquidationCollateral.add(lockedCollateral).add(
325                 ooReward
326             );
327 
328         // Construct liquidation object.
329         // Note: All dispute-related values are zeroed out until a dispute occurs. liquidationId is
                the index of the new
330         // LiquidationData that is pushed into the array, which is equal to the current length of
                the array pre-push.
331         liquidationId = liquidations[sponsor].length;
332         liquidations[sponsor].push(
333             LiquidationData({
334                 sponsor: sponsor,
335                 liquidator: msg.sender,
336                 state: Status.NotDisputed,
337                 liquidationTime: block.timestamp,
338                 tokensOutstanding: tokensLiquidated,
339                 lockedCollateral: lockedCollateral,
340                 lockedCollateralAfterWithdrawals: lockedCollateralAfterWithdrawals,
341                 disputer: address(0),
342                 settlementPrice: FixedPoint.fromUnscaledUint(0)
343             })
344         );
345 
346         // If this liquidation is a subsequent liquidation on the position, and the liquidation
                size is larger than
347         // some "griefing threshold", then re-set the liveness. This enables a liquidation against
                a withdraw request to be
348         // "dragged out" if the position is very large and liquidators need time to gather funds.
                The griefing threshold
349         // is enforced so that liquidations for trivially small # of tokens cannot drag out an
                honest sponsor's slow withdrawal.
350 
351         // We arbitrarily set the "griefing threshold" to 'minSponsorTokens' because it is the only
                parameter
352         // denominated in token currency units and we can avoid adding another parameter.
353         FixedPoint.Unsigned memory griefingThreshold = minSponsorTokens;
354         if (
355             positionToLiquidate.withdrawalRequestPassTimestamp > 0 && // The position is undergoing
                    a slow withdrawal.
356             positionToLiquidate.withdrawalRequestPassTimestamp >
357             block.timestamp && // The slow withdrawal has not yet expired.
358             tokensLiquidated.isGreaterThanOrEqual(griefingThreshold) // The liquidated token count
                    is above a "griefing threshold".
```

```
359         ) {
360             positionToLiquidate.withdrawalRequestPassTimestamp = (
361                 block.timestamp
362             ).add(withdrawalLiveness);
363         }
364
365         emit LiquidationCreated(
366             sponsor,
367             msg.sender,
368             liquidationId,
369             tokensLiquidated.rawValue,
370             lockedCollateral.rawValue,
371             lockedCollateralAfterWithdrawals.rawValue,
372             block.timestamp
373         );
374
375         // Destroy tokens
376         tokenCurrency.safeTransferFrom(
377             msg.sender,
378             address(this),
379             tokensLiquidated.rawValue
380         );
381         tokenCurrency.burn(tokensLiquidated.rawValue);
382
383         // Pull ooReward from liquidator.
384         collateralCurrency.safeTransferFrom(
385             msg.sender,
386             address(this),
387             ooReward.rawValue
388         );
389     }
```

Listing 2.21: UMA/financial-templates/expiring-multiparty/Liquidatable.sol

Suggestion I Remove the redundant check in the function `createLiquidation()`.

2.3 Notes

2.3.1 Temporary System Parameters for Testing

Status Fixed in `Version 2`

Introduced by `version 1`

Description In the contract `ClayBonds`, the parameters `APY_PERCENT`, `BONDS_ISSUANCE_PERIOD`, and `MATURATION_PERIOD` are set to a temporary value for testing convenience. All of them should be updated before the deployment.

2.3.2 Customized FinancialProductLibrary

Status Confirmed

Introduced by `version 1`

Description In the current implementation, the final settled price from the oracle will be applied by the method `_transformCollateralRequirement()` from the `FinancialProductLibrary` to get the required collateralization ratio of the contract. The method in the `FinancialProductLibrary` will currently return the [collateralRequirement which is set in the constructor. However, the creator has the ability to deploy a customized `FinancialProductLibrary`, which is beyond our audit scope.

2.3.3 Potential Centralized Problem

Status Confirmed

Introduced by `version 1`

Description The `owner` of the contract `PricelessPositionManager` has the privilege to bring forward the settlement of the contract by updating the `expirationTimestamp` of the contract via the function `emergencyShutdown()`.