Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Prime Protocol

Veridise Inc.

June 16, 2023

► **Prepared For:**

Prime Protocol Inc.
https://www.primeprotocol.xyz/

► **Prepared By:**

Ajinkya Rajput
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Jun. 16, 2023	V1.1 - Updated fix statuses
Jun. 09, 2023	V1

Contents

1 Executive Summary

From May 15, 2023 to May 26, 2023, Prime Protocol Inc. engaged Veridise to review the security of their Prime Protocol. The review was conducted as a follow-up in response to the issues discovered during a previous audit that Veridise conducted for Prime Protocol Inc. two months earlier in March 2023*. In addition to the files covered in the previous audit, this audit also covers the smart contract source code that was not in scope in the last audit, which includes the oracle module, the treasury module, and the staking module. Veridise auditors were also specifically requested to prioritize a security assessment of the rebasing logic support, which was related to many of the issues uncovered during the previous audit. Veridise conducted the assessment over 4 person-weeks, with 2 engineers reviewing code over 2 weeks on Git commit `ea69944`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Prime Protocol developers provided the source code of the Prime Protocol contracts for review. Based on the Veridise auditors' assessment, the code is largely unchanged compared to the version audited previously. We refer the reader to the previous audit report for the code assessment.

Summary of issues detected. The audit uncovered 22 issues, 1 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, inconsistent unit conversions may cause liquidators to receive fewer of the seized tokens than they should have (V-PRI3-VUL-001). The Veridise auditors also identified several medium- and low-severity issues, including a reentrancy attack vector in the StakingPool component (V-PRI3-VUL-002) and a potential subtraction overflow when updating interest rates (V-PRI3-VUL-009). Additionally, the auditors reported 9 warnings and 3 informational issues.

The Prime Protocol developers resolved all issues.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Prime Protocol. First, we recommend that the Prime Protocol developers address the recommendations in the previous audit report. Second, as this audit discovered more issues related to rebasing tokens, including the 1 high-severity issue, we recommend that the Prime Protocol developers ensure that rebasing tokens are thoroughly tested before they are enabled on the protocol.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

* The previous audit report can be found on Veridise's website at `https://veridise.com/veridise-audits/`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Prime Protocol	`ea69944`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 15 - May 26, 2023	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	1	1
Medium-Severity Issues	1	1
Low-Severity Issues	8	8
Warning-Severity Issues	9	9
Informational-Severity Issues	3	3
TOTAL	22	22

Table 2.4: Category Breakdown.

Name	Number
Logic Error	9
Maintainability	7
Data Validation	4
Reentrancy	1
Arithmetic Overflow	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Prime Protocol's smart contracts. In our audit, we sought to answer the following questions:

- Do all user flows correctly convert between ptokens and underlying tokens?
- Do the oracles correctly interact with third-party code such as in Chainlink, DIA, and Redstone?
- Are there any consistency issues in the way external exchange rates are transmitted, and can they result in bugs?
- Are there any reentrancy attack vectors when performing same-chain transactions on the master chain?
- Are appropriate interest amounts accumulated at the right times?
- Do rebasing PToken contracts correctly account for rebases?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the Solidity source files in the `contracts` folder of the source code provided by the Prime Protocol developers, which contains the smart contract implementation of the Prime Protocol. One folder/component, `Wormhole.sol`, was excluded from the scope of the audit as the Prime Protocol developers indicated that that component has to be reimplemented.

Methodology. Veridise auditors reviewed the reports of previous audits for Prime Protocol, inspected the provided tests, and read the Prime Protocol documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Prime Protocol developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-PRI3-VUL-001	Inconsistency in liquidation seize token units	High	Fixed
V-PRI3-VUL-002	Reentrancy attack vector in claimRewards with c. .	Medium	Fixed
V-PRI3-VUL-003	Underlying disbursement uses stale external exc. . .	Low	Acknowledged
V-PRI3-VUL-004	Inconsistent borrow asset exchange rate for USP. . .	Low	Fixed
V-PRI3-VUL-005	Missing gas limit checks for liquidation requests	Low	Fixed
V-PRI3-VUL-006	Missing call to _exitLoanMarket in liquidation	Low	Fixed
V-PRI3-VUL-007	PTokenBase and AavePToken send ptoken amount	Low	Fixed
V-PRI3-VUL-008	Incorrect data validation in modifyLoanAsset	Low	Fixed
V-PRI3-VUL-009	Potential subtraction overflow in setBorrowRate. . .	Low	Fixed
V-PRI3-VUL-010	User flows do not update accrued interest for c. . .	Low	Intended Behavior
V-PRI3-VUL-011	Check for USP underlying does not check chain ID	Warning	Fixed
V-PRI3-VUL-012	Extra conversion to ptokens in in completeWith. . .	Warning	Fixed
V-PRI3-VUL-013	Redundant comparison against 0	Warning	Fixed
V-PRI3-VUL-014	Contradictory comments on RequestController.wit	Warning	Fixed
V-PRI3-VUL-015	Potentially missing special case for USP in get. . .	Warning	Intended Behavior
V-PRI3-VUL-016	_syncUserRewards always applies to msg.sender	Warning	Fixed
V-PRI3-VUL-017	MAX_SIZE can be defined in terms of other const.	Warning	Acknowledged
V-PRI3-VUL-018	setAssetKey does not check whether feed exists	Warning	Acknowledged
V-PRI3-VUL-019	Price bounds are magic constants	Warning	Fixed
V-PRI3-VUL-020	Unused mappings in PrimeOracleStorage	Info	Fixed
V-PRI3-VUL-021	Confusing naming in supportSatelliteLoanMarket	Info	Fixed
V-PRI3-VUL-022	Inconsistent naming in TreasuryAdmin methods	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-PRI3-VUL-001: Inconsistency in liquidation seize token units

Severity	High	**Commit**	ea69944
Type	Logic Error	**Status**	Fixed
File(s)	`MasterInternals.sol`, `RebasePTokenMessageHandler.sol`		
Location(s)	_liquidateBorrow(), seize()		

Liquidation requests are routed to `MasterInternals._liquidateBorrow()`, which will determine whether the liquidation should proceed and how much of the liquidated collateral ("seize tokens") to reward to the liquidator. If the liquidation is approved, the master chain will send a message to the satellite chain of the seized collateral to grant the collateral to the liquidator. However, the units for the seized token amount are inconsistent between the master and satellite.

There are three places that exhibit the inconsistency:

1. The seize token amount is calculated in `_liqudiateCalculateSeizeTokens()`, where the returned `seizeAmount` is in terms of underlying tokens for that collateral. This `seizeAmount` is the number that will be sent over to the satellite to reward to the liquidator.
2. The `seizeAmount` is deducted from the `markets[..][..].totalSupply` field in `_liquidateBorrow()`, but the `totalSupply` field is denominated in terms of ptokens.

```
markets[seizeMarket.chainId][seizeMarket.asset].totalSupply -= seizeAmount;
```

Snippet 4.1: Location in `liquidateBorrow()` where the `totalSupply` field is adjusted.

3. The `RebasePTokenMessageHandler.seize()` and the `PTokenMessageHandler.seize()` methods assume that the `seizeTokens` field of the `SLiquidateBorrow` packet is in terms of ptokens. This is inconsistent with the units of `seizeAmount`, which should be in terms of underlying. Furthermore, when underlying tokens are transferred to the liquidator, the ptoken amount is transferred, not the underlying amount. Note that the `seize()` method of

```
function seize(
    IHelper.SLiquidateBorrow memory params
) external payable virtual override onlyMid() {
    if (isFrozen) revert MarketIsFrozen(address(this));

    totalSupply -= params.seizeTokens;

    uint256 actualSeizeTokens = (params.seizeTokens * params.externalExchangeRate) /
    10**EXCHANGE_RATE_DECIMALS;

    _doTransferOut(params.liquidator, underlying, actualSeizeTokens);
}
```

Snippet 4.2: Implementation of `RebasePTokenMessageHandler.seize()`; the implementation in `PTokenMessageHandler` is similar.

`CompoundPTokenMessageHandler`, unlike in the other "message handler" contracts, assumes that the `seizeTokens` field is in terms of underlying.

Impact As mentioned above, the inconsistencies are related to the liquidation user flow. These have the following effects:

- When the external exchange rate is not 1:1, then an incorrect amount of seize tokens will be transferred to liquidators. This typically applies to `RebasePToken` implementations, in which case:
  - An external exchange rate above one underlying-per-ptoken may lead to a denial of service issue in `seize()`, as it is possible for `totalSupply` to be smaller than `params.seizeTokens` or for the `actualSeizeTokens` to exceed the actual balance of the contract.
  - An external exchange rate below one underlying-per-ptoken may lead to an underpayment issue. The protocol will considered the whole seize amount to be transferred, but in reality, only a fraction of the intended seize token amount will be transferred to the liquidator. The remaining amount that should've been transferred will instead be "given" to the remaining depositors of the PToken.
- For money market PTokens, the `seizeAmount` deducted from the `.totalSupply` field in `_liquidateBorrow()` will not include the accumulated interest. Thus, the deducted `seizeAmount` will only partially account for the amount of seized collateral removed from the protocol. Consequently, `.totalSupply` (which is denominated in ptokens) will be higher than it should be; this will bias future underlying-per-ptoken exchange rates to favor the underlying side.

Recommendation The developer should clarify the expected units of the seize token amounts and resolve the inconsistencies.

Developer Response The developers stated that the `seizeAmount` should be denominated in underlying tokens, and they have changed the implementation of `RebasePTokenMessageHandler.seize()`. They also confirmed the issue with the `markets[...][...].totalSupply` subtraction and have changed the calculation so that the `_exchangeRate()` is first applied to the `seizeAmount`.

4.1.2 V-PRI3-VUL-002: Reentrancy attack vector in claimRewards with callback-supporting ERC20s

Severity	Medium	Commit	ea69944
Type	Reentrancy	Status	Fixed
File(s)	StakingPool.sol		
Location(s)	claimRewards()		

The `claimRewards()` function allows a staker to claim accrued reward tokens (which are ERC20 tokens). The accumulated reward balance is maintained in the `userRewards` mapping. The

```
function claimRewards(address rewardsToken, uint256 rewardsAmount) external {
    _syncRewardsMarkets();
    _syncUserReward(rewardsToken);
    uint256 userRewardsBalance = userRewards[msg.sender][rewardsToken].
    accumulatedRewards;
    if (rewardsAmount == type(uint256).max) {
        rewardsAmount = userRewardsBalance;
    }
    if(userRewardsBalance < rewardsAmount) revert InsufficientRewards();
    if(rewardsAmount > 0){
        _doTransferOut(msg.sender, rewardsToken, rewardsAmount);
        userRewards[msg.sender][rewardsToken].accumulatedRewards -= rewardsAmount;
        emit RewardsClaimed(msg.sender, rewardsToken, rewardsAmount);
    }
}
```

Snippet 4.3: Implementation of `StakingPool.claimRewards()`

function first calculates the `rewardsAmount` being claimed and checks that there are sufficient accrued rewards to grant. Next, if `rewardsAmount` is greater than zero, the rewards are first transferred out, and then the balance in `userRewards` mapping is deducted by the rewards amount.

If the reward token supports callbacks, then the ordering of the operations makes the function vulnerable to a reentrancy attack.

Impact If the reward token supports callbacks and the staker is a malicious smart contract, then the staker can make a reentering call into `claimRewards()`. Note that because the `userRewards` entry is not updated until after the transfer is finished executing, this means that the staker will be granted additional reward tokens a second time. The staker can reenter into the `claimRewards()` function multiple times to obtain multiples of the amount that they wanted to claim.

Recommendation To reduce the risk of a reentrancy attack, the `userRewards` entry should be updated before the transfer occurs.

4.1.3 V-PRI3-VUL-003: Underlying disbursement uses stale external exchange rate

Severity	Low	**Commit**	ea69944
Type	Logic Error	**Status**	Acknowledged
File(s)	`PTokenMessageHandler.sol`, `RebasePTokenMessageHandler.sol`, `AavePTokenMessageHandler.sol`		
Location(s)	completeWithdraw(), seize()		

A rebasing token is handled by the `RebasePToken` contract, which has special logic to account for changes to the contract's token balance as a result of a rebase. Specifically, whenever a user deposits the underlying rebasing token, they will be granted a proportional amount of non-rebasing "ptokens". The user can then use the withdraw flow to redeem ptokens for the underlying token.

As part of the conversion between ptokens and underlying, the protocol tracks an "external exchange rate" for the underlying token, which is the ratio of the underlying token balance to the amount of ptokens. For example, if there are 100 ptokens and 100 underlying in the PToken contract, then the external exchange rate is 1 (i.e., 1:1). If a rebase occurs and the underlying balance increases to 120, then the external exchange rate is 1.2 underlying-per-ptoken. Based on

```
1  function _getExternalExchangeRate() internal virtual override returns (uint256
       externalExchangeRate) {
2      if (totalSupply == 0) {
3          externalExchangeRate = 10**EXCHANGE_RATE_DECIMALS;
4      } else {
5          IERC20 token = IERC20(underlying);
6          uint256 cash = token.balanceOf(address(this));
7          externalExchangeRate = (cash * 10**EXCHANGE_RATE_DECIMALS) / totalSupply;
8      }
9
10     if (currentExchangeRate != externalExchangeRate) currentExchangeRate =
       externalExchangeRate;
11 }
```

Snippet 4.4: Definition of `_getExternalExchangeRate()`, which computes the external exchange rate.

the way that the external exchange rate is used in the protocol, withdrawals and liquidations are subject to consistency issues caused by replication lag, in a distributed systems sense:

- **Satellite replicates external exchange rate to master on deposits**. During the deposit flow, the user-provided underlying amount is converted into ptokens using the current external exchange rate. Both the underlying amount and the external exchange rate will be sent to the master state, which will update the bookkeeping. Notably, the master state will record the external exchange rate it received from the satellite.
- **Master uses the replicated external exchange rate for withdrawals and liquidations**. During the withdraw flow, a user will send a withdraw request to master state (potentially from a satellite chain different from that of the PToken being withdrawn from), where they will indicate the amount of ptokens to withdraw. The master state will use the previously recorded external exchange rate to calculate the amount of underlying to credit to recipient, and then it send the underlying amount and the previously recorded


```
uint256 externalExchangeRate = _getExternalExchangeRate();
uint256 actualTransferAmount = _doTransferIn(underlying, user, amount);
uint256 actualDepositAmount = (actualTransferAmount * 10**EXCHANGE_RATE_DECIMALS) /
    externalExchangeRate;

_sendDeposit(
    route,
    user,
    underlying == address(0)
        ? msg.value - actualTransferAmount
        : msg.value,
    actualTransferAmount,
    externalExchangeRate
);

totalSupply += actualDepositAmount;
```

Snippet 4.5: Lines in `RebasePToken.depositBehalf()` that will send the current external exchange rate to the master state.

external exchange rate to the satellite chain of the PToken. This underlying amount will be transferred to the recipient, and the PToken will update its bookkeeping based on the previously recorded external exchange rate. In the liquidation flow, the liquidator will be granted the liquidated collateral (called the "seize tokens"). The seize token amount is calculated similar to how it is done in the withdraw flow.

- **Master uses the replicated external exchange rate for pricing**. The master state records collateral balances as ptokens, not as underlying. In order to calculate the US$ price of the collateral, it needs to compute the equivalent underlying amount of each ptoken using the external exchange rate.

Note that because external exchange rate is only updated on master during a deposit, this means that the calculation of the underlying amount for withdrawals, liquidations, and pricing may be based on a stale exchange rate.

Impact on withdrawals and liquidations Using a stale external exchange rate can lead to at least two problems, including overdrafting and locked funds. As one concrete example, consider the following scenario where overdrafting arises when the rebasing token reduces token balances:

1. Assume initially that there is a `RebasePToken` contract for a rebasing token (underlying), where the current external exchange rate is 1:1. Further assume that the `RebasePToken` and the underlying token are both located on a satellite chain.
2. Suppose some user Alice has no ptokens and no deposited collateral, the underlying balance of the `RebasePToken` is `X`, and the total supply of ptokens is `X` (they are equal because the external exchange rate is 1:1).
3. Alice deposits 100 underlying tokens into the `RebasePToken` contract; since the current exchange rate is 1, she receives 100 ptokens. The new underlying balance and ptoken total supply of the `RebasePToken` are both `X + 100`. After the master state receives the deposit message, it records the ptoken balance as `X + 100` and the external exchange rate as 1.

```
function completeWithdraw(
    IHelper.FBWithdraw memory params
) external payable virtual override onlyMid() {
    if (isFrozen) revert MarketIsFrozen(address(this));

    emit WithdrawApproved(
        params.user,
        address(this),
        params.withdrawAmount,
        true
    );

    uint256 pTokenWithdrawAmount = (params.withdrawAmount * 10**
    EXCHANGE_RATE_DECIMALS) / params.externalExchangeRate;

    totalSupply -= pTokenWithdrawAmount;

    _doTransferOut(params.user, underlying, params.withdrawAmount);
}
```

Snippet 4.6: Definition of `RebasePTokenMessageHandler.completeWithdraw()`. Note that the `params.withdrawAmount` is in terms of underlying, and that the `params.externalExchangeRate` is the one sent from the master state.

```
        (uint256 exchangeRate,) = _exchangeRate(pToken, pTokenChainId);
    // Pre-compute a conversion factor from tokens -> usp (should be 1e18)
    tokensToDenom = exchangeRate * collateralFactor * oraclePrice / 10**
    factorDecimals / 10**oracleDecimals;
}

uint256 pTokenDecimals = 10**markets[pTokenChainId][pToken].decimals;
uint256 collateralValue;
{
    // Note: We don't use '_collateralBalanceStored()' here since the exchangeRate
    has already been applied in 'tokensToDenom'
    uint256 collBal = pTokenCollateralBalances[pTokenChainId][user][pToken];

    collateralValue = tokensToDenom * collBal / pTokenDecimals;
}
```

Snippet 4.7: Relevant lines in `MasterInternals._getValueOfCollateral()` for pricing. If the `pToken` is a rebasing PToken, then the result of `_exchangeRate()` will be equal to the previously saved external exchange rate. The user's ptoken (collateral) balance will be multiplied by the saved external exchange rate to convert it to underlying tokens, which will then be multiplied by the oracle price to obtain the US$ value.

4. The rebasing token triggers a rebase so that each account has only 80% of its original token balance. The balance of the `RebasePToken` is now `0.8 * X + 80`; the ptoken total supply remains as `X + 100`.
5. On a satellite chain different from that of the `RebasePToken`, Alice sends a withdraw request of 100 ptokens to the master state. The master state approves; because the last recorded external exchange rate is 1, the approved withdraw amount is 100 underlying tokens.
6. When the satellite chain receives the response, it calls `RebasePToken.completeWithdraw()`. This will deduct the original amount of ptokens requested from the `totalSupply` of ptokens, so that the new ptoken total supply is `X`. For underlying, there are two possible outcomes:
    a) If the current underlying balance `0.8 * X + 80` is less than 100, then the function will revert as the contract will attempt to transfer 100 underlying to the recipient, even though the underlying balance is less than that.
    b) Otherwise, 100 underlying will be transferred to the recipient. The external exchange rate before the withdraw is `(0.8 * X + 80) / (X + 100) = 0.8`, so the underlying amount corresponding to Alice's requested withdrawal of 100 ptokens is only 80 underlying tokens. Conceptually, this means that the actual transferred amount is larger than what should have been transferred, as it will consist of both Alice's original deposit of 80 underlying as well as an excess 20 underlying that is drawn from other depositors of the contract.
    Furthermore, the external exchange rate after the withdraw will be `(0.8 * X - 20) / X = 0.8 - 20 / X`. As the rate is now lower, any following deposits will grant a higher amount of ptokens for each underlying token. For example, Alice can decide to deposit back the 100 underlying tokens and theoretically receive up to 167 ptokens (depending on how small `X` is), which is much larger than the 100 ptokens that Alice had started with. This would then lock in the external exchange rate, to the detriment of existing depositors (who would not gain as much underlying per ptoken on withdrawals as Alice did).

If we take the above scenario and consider when the rebasing token increases token balances, then the situation is a locked funds issue: **less** underlying will be transferred compared to what should have been transferred, so that some amount of underlying will effectively be transferred to all other depositors of the PToken contract.

The issues discussed above also occur for tokens seized during a liquidation.

Impact on borrowing The external exchange rate is directly proportional to the calculated US$ of collateral. Thus, a stale external exchange may cause the collateral to be overvalued or undervalued by the master state. If the collateral is overvalued by the master state (current rate is lower than last recorded rate), this may lead to loans being approved when a user uses a rebasing token as collateral when the collateral is technically insufficient for the loan. The next deposit would update the external exchange rate and then cause the user to become liquidatable. In contrast, if the collateral is undervalued by the master state, than the user will be required to provide more collateral than may be required.

Recommendation Resolving the pricing issue would seem to require large changes to the protocol, so we leave decisions regarding the pricing issue to the discretion of the developers.

For the withdraw and liquidation issues, we recommend that the developers either address the consistency issue in the external exchange rate, or provide mitigations in the event that the external exchange rate is inconsistent. Some potential mitigations, each with different tradeoffs, include:

- Calculate the underlying amount on the PToken side using the current external exchange rate instead of calculating it on the master state with the stale external exchange rate. This can help avoid consistency issues with withdrawals and liquidations.
- Change the withdraw & liquidation flows to use an atomic commitment protocol, such that a withdraw or liquidate transaction will be aborted if there is a change in the external exchange rate, an oracle price update, or other side effect on the same account while the transaction is being executed. This will prevent stale exchange rates from being used; however it will introduce significant complexity to the protocol and will increase latency during withdrawals and liquidations.
- Force withdraw & liquidation requests to be first routed through the satellite chain of the corresponding PToken, so that it can attach the current external exchange rate to the withdraw/liquidation request before sending it to the master chain. This strategy will likely cause the external exchange rate to be up-to-date if no deposits have occurred in a long time, but it will not correctly handle a rebase that occurs after the withdraw/liquidation request is sent to but not yet received by the master state. Again, it will also incur additional latency as an extra message may need to be sent.

Developer Response The developers noted that they only plan to use rebasing tokens whose supply can only increase. They also noted that they expect these rebasing tokens to be interest-bearing tokens, so a stale exchange rate will cause the depositor to forfeit their accrued interest to the other depositors. Lastly, they indicated that the interest rates are expected to be low, so the effect will likely be small.

4.1.4 V-PRI3-VUL-004: Inconsistent borrow asset exchange rate for USP underlying

Severity	Low	Commit	ea69944
Type	Logic Error	Status	Fixed
File(s)	PrimeOracle.sol		
Location(s)	getBorrowAssetExchangeRate()		

The `PrimeOracle.getBorrowAssetExchangeRate()` method is used to calculate an exchange ratio between two assets, "overlying" and "underlying". In the general case, this divides the oracle price of the overlying by the oracle price of the underlying. However, when the underlying is the Prime protocol's USP token, then the method directly returns the US$ price of USP from the oracle. Overlying is not considered at all, which seems strange. Furthermore, the units in these two cases are different—one is a ratio between overlying and underlying, and the other has US$ per underlying.

```
function getBorrowAssetExchangeRate(
    address overlying,
    uint256 overlyingChainId,
    address underlying,
    uint256 underlyingChainId
) external view override returns (uint256 ratio, uint8 decimals) {
    if (uspAddress == address(0)) revert UspAddressZero();

    if (underlying == uspAddress) {
        return _getAssetPrice(block.chainid, underlying);
    }

    (uint256 numAnswer, uint8  numDecimals) = _getAssetPrice(overlyingChainId,
    overlying);
    (uint256 denAnswer, uint8  denDecimals) = _getAssetPrice(underlyingChainId,
    underlying);

    if (numAnswer >= 0 && denAnswer >= 0) {
        ratio = numAnswer * 10**(denDecimals + RATIO_DECIMALS) / denAnswer / 10**
    numDecimals;
    }

    decimals = RATIO_DECIMALS;
}
```

Snippet 4.8: Implementation of `getBorrowAssetExchangeRate()`

Impact `getBorrowAssetExchangeRate()` is used in several places:

- to calculate collateral amounts seized during liquidations in `MasterInternals._liquidateCalculateSeizeTok()`; the underlying is the collateral being seized
- to calculate loan market premiums in `getLoanMarketPremium()`, where the underlying is the loan asset's underlying
- for setting the borrow rate in `StairIRM.setBorrowRate()`, where the underlying is the loan asset's underlying

If USP is the underlying in any of these cases, incorrect values may be computed.

Recommendation The developers should clarify the intended behavior of the USP underlying case and fix the implementation to match the intended behavior.

Developer Response The developers changed the method so that it will "return price of overlying if underlying is USP, since USP price is always 1 for the protocol as well as set numerator to 1 if USP is underlying." The auditors noted that the fix does not account for the case where both are USP; however, the developers stated that there will be no situation in which both overlying and underlying are USP.

4.1.5 V-PRI3-VUL-005: Missing gas limit checks for liquidation requests

Severity	Low	**Commit**	ea69944
Type	Data Validation	**Status**	Fixed
File(s)	`RequestController.sol`		
Location(s)	liquidate()		

The Prime developers recently added gas limit checks to the deposit and repay user flows in `RequestController` to ensure that users supply sufficient gas. Based on a discussion with the developers, this was done to prevent flows that require user funds from getting "stuck" if insufficient gas is supplied. Such validation is present in the `deposit()`, `repayBorrow()` and `repayBorrowBehalf()` methods, but not in `liquidate()`. Liquidation requires the liquidator to repay the loan on the satellite before a message is sent to the master.

```
function repayBorrowBehalf(
    address borrower,
    address route,
    address loanMarketAsset,
    uint256 repayAmount
) external payable virtual override returns (uint256) {
    if (block.chainid == masterCID && uint256(gasleft()) < uint256(550000)) revert
    GasLimitTooLow(uint256(gasleft()));
    if (repayAmount == 0) revert ExpectedRepayAmount();
```

Snippet 4.9: Example of the gas limit check in `repayBorrowBehalf()`

Impact Liquidators may run into the same gas issue (and "stuck" transactions) that depositors and borrowers ran into.

Recommendation Add a similar gas limit check for liquidations. Based on a discussion with the developers, the gas limit may need to be higher for liquidations as it involves more logic.

4.1.6 V-PRI3-VUL-006: Missing call to _exitLoanMarket in liquidation

Severity	Low	**Commit**	ea69944
Type	Logic Error	**Status**	Fixed
File(s)	`MasterInternals.sol`		
Location(s)	_liquidateBorrow()		

A liquidator can request to liquidate an insolvent borrower by repaying the borrow's loan on the satellite chain of that loaned asset and specifying which of the borrower's collateral they would like to seize in return. However, there is a discrepancy between the repay user flow's handling in `_repay()` and the liquidation user flow's handling in `_liquidateBorrow()`. The repay flow will invoke `_exitLoanMarket()` if the repayer pays off the loan entirely, whereas the liquidation flow does not appear to have any similar code.

```
if (accountLoanMarketBorrows[borrower][targetMarket.loanAsset][targetMarket.chainId].
    principal == 0) {
    if (!_exitLoanMarket(
        borrower,
        targetMarket
    )) revert ExitLoanMarketFailed();
}
```

Snippet 4.10: The snippet of code in `_repay()` that appears to be missing in `_liquidateBorrow()`.

Impact This issue currently does not appear to have any impact other than higher gas consumption, but it may cause potential problems if the developers want to extend or modify the code.

The `_exitLoanMarket()` affects two storage variables: `isLoanMarketMember` and `accountLoanMarkets`. The former indicates whether an account has taken borrowed any funds from a loan market, and it is only used in `_enterLoanMarket()`. The latter tracks the loan markets that an account has borrowed from, so that the protocol can track the US$ value of collateral and loaned assets. The missing call to `_exitLoanMarket()` will not have any effect on those calculations, as the principal amount will be zero in this scenario.

Recommendation The developers should update `_liquidateBorrow()` with a call to `_exitLoanMarket()` if the final principal amount is zero.

4.1.7 V-PRI3-VUL-007: PTokenBase and AavePToken send ptoken amounts instead of underlying

Severity	Low	Commit	ea69944
Type	Logic Error	Status	Fixed
File(s)	PTokenBase.sol, AavePToken.sol		
Location(s)	depositBehalf()		

The `depositBehalf()` method is used to invoke the deposit user flow. This requires sending a message to the master state with the number of underlying tokens that have been deposited by the user, along cross-chain transaction gas fee (in `msg.value`). However, the actual amount of underlying tokens is in `actualTransferAmount`, whereas the amount transferred over is the equivalent ptoken amount `actualDepositAmount`. Furthermore, if the underlying token is the native currency, then the ptoken amount will be deducted from the gas fee, not the native currency amount. In comparison, the `RebasePToken` implementation correctly sends over

```
function depositBehalf(
    address route,
    address user,
    uint256 amount
) public virtual override payable sanityDeposit(amount, user) {
    uint256 externalExchangeRate = _getExternalExchangeRate();
    uint256 actualTransferAmount = _doTransferIn(underlying, user, amount);
    uint256 actualDepositAmount = (actualTransferAmount * 10**EXCHANGE_RATE_DECIMALS)
     / externalExchangeRate;

    _sendDeposit(
        route,
        user,
        underlying == address(0)
            ? msg.value - actualDepositAmount
            : msg.value,
        actualDepositAmount,
        externalExchangeRate
    );
}
```

Snippet 4.11: Implementation of `depositBehalf()`

underlying tokens. The `CompoundPToken` does not use ptokens in its implementation.

Impact The default implementation of `PTokenBase._getExternalExchangeRate()` returns 1 (with the correct precision), so the external exchange rate of `PTokenBase` and its subclass `AavePToken` will always be 1. Thus, this bug currently has no impact, but the bug may be triggered if the developers decide to add a PToken that has a non-1:1 exchange rate that extends from `PTokenBase`. In this situation, the amount sent over will be ptokens and not underlying, which may result in a discrepancy in the bookkeeping of the protocol.

Recommendation Change `actualDepositAmount` to `actualTransferAmount` in all occurrences in the call to `_sendDeposit()`.

4.1.8 V-PRI3-VUL-008: Incorrect data validation in modifyLoanAsset

Severity	Low	**Commit**	ea69944
Type	Data Validation	**Status**	Fixed
File(s)	`TreasuryAdmin.sol`		
Location(s)	modifyLoanAsset()		

The `modifyLoanAsset()` method suffers from several data validation issues:

1. There is no check that for whether the trade asset is actually supported, i.e. whether `supportedTradeAssets[_localLoanAsset][_tradeAsset]` is true.
2. The function appears to support a `_mintPrice` and `_burnPrice` parameter of 0, corresponding to a no-op; however, the bounds check does not allow the `_mintPrice` and `_burnPrice` to be zero. They are restricted to the range described in the comments.

```
function modifyTradeAsset(
    address _localLoanAsset,
    address _tradeAsset,
    uint256 _mintPrice,
    uint256 _burnPrice
) external onlyAdmin() {
    unchecked {
        /* Min: 1e8 */
        /* Max: 105e6 */
        if (_mintPrice - 1e8 > 5e6) revert ParamOutOfBounds();
        if (_burnPrice - 1e8 > 5e6) revert ParamOutOfBounds();
    }
    if (_mintPrice != 0) localLoanAsset[_localLoanAsset][_tradeAsset].mintPrice =
    _mintPrice;
    if (_burnPrice != 0) localLoanAsset[_localLoanAsset][_tradeAsset].burnPrice =
    _burnPrice;
}
```

Snippet 4.12: Implementation of `modifyLoanAsset()`

Impact

1. Without the check for a valid trade asset / loan asset pair, it may be possible for the admin to accidentally call `modifyTradeAsset` on an invalid pair.
2. An admin will be required to provide both `_mintPrice` and `_burnPrice` to `modifyTradeAsset()`. If the admin only intends to set one of the prices, they will still be required to provide both. This may be error prone, as they will have to look up the existing price if they do not want to change it.

Recommendation First, the developers should insert a check that the trade asset / loan asset pair is supported. Second, the bounds checks should be pushed into the branches, so that the corresponding check only occurs if the price is nonzero. Lastly, to improve readability, we recommend that the developers indicate that the price parameters can be set to zero.

4.1.9 V-PRI3-VUL-009: Potential subtraction overflow in setBorrowRate branching logic

Severity	Low	Commit	ea69944
Type	Arithmetic Overflow	Status	Fixed
File(s)	StairIRM.sol		
Location(s)	setBorrowRate()		

The `StairIRM` is an interest rate model such that whenever the interest rate falls outside of a "stable range", it will be adjusted back towards the stable range by a fixed discrete amount (the `basisPointsTickSize`) at regular intervals. This adjustment is performed by the `setBorrowRate()` method, where it handles the adjustment in a chain of if-else statements. Two of these cases may be prone to integer overflow:

1. Subtraction overflow will occur if the `basisPointsTickSize` is greater than the `basisPointsUpperTick` .

```
// decrease interest below stable range
else if (ratio < lowerTargetRatio) {
    if (_borrowInterestRatePerBlock <= (basisPointsUpperTick -
    basisPointsTickSize)) {
        _borrowInterestRatePerBlock += basisPointsTickSize;
    }
```

2. Subtraction overflow will occur if the `basisPointsTickSize` is greater than the `_borrowInterestRatePerBlock` .

```
//tick back towards baseline inside stable range
} else if (_borrowInterestRatePerBlock > borrowInterestRateBaseline) {
    // avoid oscillation below baseline
    if (_borrowInterestRatePerBlock - basisPointsTickSize <
    borrowInterestRateBaseline)
        _borrowInterestRatePerBlock = borrowInterestRateBase;
    else {
```

Impact If subtraction overflow occurs in `setBorrowRate()`, then the transaction will revert. The `setBorrowRate()` method is used by `MasterInternals._accrueInterestOnSingleLoanMarket()`, which is called by all of the user flows except for the deposit user flow. Specifically, this will lead to a denial-of-service issue when all of the conditions are true:

- The affected `StairIRM` is the interest rate model for a loan market (which we will call the "affected" loan market);
- There is an account (which we will call the "affected account") participating in that loan market (i.e., they have an outstanding loan for that market).
- And one of the following is true:
  - The affected account is the withdrawer in the withdraw user flow.
  - The affected account in the borrower in the borrow user flow, and the target market to borrow from is the affected loan market.
  - The affected account is the repayer in the repay user flow, and the target market being repaid is the affected loan market.

- The affected account is the borrower in the liquidation user flow, and the target market being repaid is the affected loan market.

The issue is exacerbated by V-PRI3-VUL-006.

Recommendation To avoid subtraction overflow, the subtraction of `basisPointsTickSize` can be converted into an addition on the other side of the comparison.

4.1.10 V-PRI3-VUL-010: User flows do not update accrued interest for collateral

Severity	Low	**Commit**	ea69944
Type	Logic Error	**Status**	Intended Behavior
File(s)	`MasterInternals.sol`		
Location(s)	See description		

In all user flows, interest is not accrued for money market assets used as collateral unless borrows, repayments, or liquidations are occurring on those same money market assets. This affects all user flows. The root cause of this issue is described below.

First, the bookkeeping for a collateral market keeps the "ptokens" in the `markets[][].totalSupply` field and the amount of interest-accumulated amount of underlying tokens in the `loanMarkets[][].totalSupplied` field. In the context of a collateral market, a ptoken is like a "share", where they can be redeemed for the principal amount of underlying token plus accumulated interest.

A money market asset is represented by a `PToken` contract on a satellite chain. On the master state, underlying amounts are converted to ptoken amounts (and vice versa) using the `_exchangeRate()`, denominated in terms of underlying-per-ptoken. Note that the underlying amount in the `_exchangeRate()` includes the principal *plus the interest*. Lastly, the withdraw, borrow, repay, and

```
// Lines in _exchangeRate() where the exchange is calculated for a money market asset
    .
uint256 totalSupply = markets[pTokenChainId][pToken].totalSupply;
uint256 totalSupplied = /* irrelevant */;
{
    LoanMarketMetadata memory targetMarket = mappedLoanAssets[pTokenChainId][pToken];
    if (targetMarket.chainId != 0) {
        totalSupplied = loanMarkets[targetMarket.loanAsset][targetMarket.chainId].
    totalSupplied * normalizeFactor;
    }
}
if (totalSupplied == 0 ||  totalSupplied == type(uint256).max) return (
    normalizeFactor, normalizeFactor);
uint256 exchangeRate = totalSupplied / totalSupply;
return (exchangeRate, normalizeFactor);
```

Snippet 4.13: Relevant lines in `_exchangeRate()`. For a money market asset, the `mappedLoanAssets` entry will be defined, so the final exchange rate is the `.totalSupplied` of the loan market divided by the `.totalSupply` of the collateral market.

liquidate user flows all start by calling the internal functions `_accrueInterestOnAllLoanMarkets()` and `_accrueInterestOnSingleLoanMarket()`. Given an account, these two internal functions update the accrued interest amounts for the loan markets that the account has borrowed from. Specifically, interest from the total borrowed amount `.totalBorrows` is accumulated into the `.totalSupplied` that is available to be loaned out. However, we note that these functions are only ever called on amounts being loaned/borrowed, and never for collateral amounts.

Impact Ultimately, this means that money market assets used as a collateral will be undervalued if there is little borrow, repay, or liquidation action for those money market assets. Specifically,

```
function _accrueInterestOnSingleLoanMarket(
    LoanMarketMetadata memory targetMarket
) public virtual override {
    LoanMarket memory loanMarket = loanMarkets[targetMarket.loanAsset][targetMarket.
    chainId];

    if (loanMarket.accrualBlockNumber != block.number) {
      /* ... */
      uint256 interestAccumulated;
            {
          /* ... */
          interestAccumulated = simpleInterestFactor * loanMarket.totalBorrows /
    normalizeFactor;
          /* ... */
      }
      /* ... */
            loanMarket.totalBorrows += interestAccumulated;
            {
                uint256 adminFee = loanMarket.adminFee;
                uint256 feeNormalizeFactor = 10**FEE_PRECISION;
                if (loanMarket.totalSupplied != type(uint256).max && adminFee !=
    feeNormalizeFactor) {
                    uint256 supplyInterestAccumulated = interestAccumulated * (
    feeNormalizeFactor - adminFee) / feeNormalizeFactor;
                    loanMarket.totalSupplied += supplyInterestAccumulated;
                }
```

Snippet 4.14: Relevant lines in `_accrueInterestOnSingleLoanMarket()` that update the accrued interest amounts.

missing interest accruals on collateral will result in the following effects:

- Because the `.totalSupplied` is in the numerator when calculating the `_exchangeRate()` for a money market asset, a missing update to the accrued interest will result in a lower underlying-per-ptoken exchange rate than expected.
- The `_getValueOfCollateral()` method calculates the US$ value of a collateral token. The calculation involves a multiplication with the value of the `_exchangeRate()` function for that token. This means that if the accrued interest has not been updated since the last borrow or repay (directly or through a liquidation), then the US$ value will be lower than expected. Consequently, borrows may be rejected or liquidations may be approved when they should not be.
- When a user deposits (underlying) collateral into a money market, the bookkeeping for that money market asset will not include the most recently accrued interest since the last borrow or repay (directly or through a liquidation). This means that the user will receive a higher amount of ptokens for their underlying deposit than they should have, since the underlying-per-ptoken exchange rate will be lower than expected.
- When a user withdraws underlying tokens of a money market asset by supplying the corresponding ptokens, the interest of the money market asset since the last borrow or repay (directly or through a liquidation) will not be accrued. This has two effects: 1) when calculating the hypothetical liquidity of the user's account, money market assets that are used as collateral may be undervalued because the accrued interest will not be updated; 2) the disbursed underlying token amount will be lower than expected, since


```
(uint256 exchangeRate, uint256 normalizeFactor) = _exchangeRate(pToken, pTokenChainId
    );
actualDepositAmount = depositAmount * normalizeFactor / exchangeRate;
```

Snippet 4.15: Relevant lines in `_deposit()` where the ptoken amount is calculated. The `depositAmount` is the amount of underlying supplied by the user on the satellite side. The `actualDepositAmount` is the number of ptokens granted to the user for the deposit

the underlying-per-ptoken exchange rate will be lower.

```
(uint256 exchangeRate, uint256 normalizeFactor) = _exchangeRate(
    pToken,
    pTokenChainId
);

actualWithdrawAmount = pTokenWithdrawAmount * exchangeRate / normalizeFactor;
```

Snippet 4.16: Relevant lines in `_withdrawAllowed()` where the underlying amount is calculated.

- When a user's loan is liquidated and the collateral being seized is a money market asset, the interest for the money market asset will not be accrued. This means that 1) the collateral is undervalued when the borrower's account liquidity is calculated; and 2) the amount of underlying tokens seized will be lower than expected.

Theoretical Scenarios To quantify the effect of the missing interest accrual updates, we provide a few hypothetical examples below. The examples below suggest that errors are possible in theory, but negligible in practice with the parameters that the developers intend to use.

Initial parameters:

- Users in the scenario: Depositor, Borrower
- Money market collateral C1 worth 100 USD per C1.
- Lendable asset C2 worth 1 USD per C2.
- The scenario begins at zero deposit and zero borrows at time T=0
- Interest rate is 0.01 per time step. Note that this is a greatly exaggerated interest rate; if we take a "time step" to mean a block, this would correspond to about 1000% interest per year.

Protocol setup:

1. At time T = 0, Depositor deposits 100 C1. The protocol issues 100 C1 ptokens.
2. At time T = 5, Borrower borrows 50 C1. Since there are currently no borrows of C1, zero interest is accrued.
3. The remaining steps occur at time T = 100.

Exaggerated Withdrawal Scenario Suppose the Depositor attempts to withdraw 10 C1 ptokens.

- In the current implementation, the exchange rate is calculated as 1 C1 underlying per C1 ptoken. Thus, the depositor receives 10 C1 underlying tokens.

- If instead the interest for C1 is accrued at the beginning of the withdrawal, the accumulated interest is `(50 / 100) * 0.01 * (100 - 5) = 47.5` C1 tokens, bringing the loan market's `totalSupplied` to `147.5`. Thus, the exchange rate is `1.475`, so the Depositor receives 14.75 C1 tokens.

Thus, the Depositor loses out on 4.75 C1 tokens (32%) of the 14.75 C1 tokens they should have received.

Exaggerated Borrow Scenario Suppose the Depositor requests a loan worth 11000 C2.

- In the current implementation, the exchange rate is calculated as `1` C1 underlying per C1 ptoken. Thus, the total liquidity of the Depositor is `100 C1 ptokens * 1 C1 underlying / C1 ptoken * 100 C1/USD = 10000 USD`. Because the request loan amount is larger than the liquidity, the loan request is rejected.
- If instead the interest for C1 is accrued at the beginning of the withdrawal, the total loan market C1 underlying token amount after accumulating interest is `147.5` C1 tokens. The exchange rate is `1.475`, so the total liquidity of the Depositor is `100 C1 ptokens * 1.475 C1 underlying / C1 ptoken * 100 C1/USD = 14750 USD`. The liquidity is larger than the requested loan amount, so the loan request is granted.

Here, the Depositor only has 68% of the liquidity that they should have at the time of the borrow.

Semi-Realistic Withdraw Scenario Lastly, we considered a semi-realistic scenario involving a modified version of the protocol setup above:

- We take a "time step" to mean a block. Assuming 7000 blocks per year and 10% APY, we calculate the interest rate per block to be $3.914 * 10^{-8}$ per block (as a factor, not as a percentage).
- The initial amount of C1 deposited into the protocol is `999000` C1 ptokens, 1 million C1 underlying tokens, and `600000` C1 underlying tokens borrowed.

Then suppose the following occurs (we omit the calculations for brevity):

1. At time T = 0, the Depositor deposits 100 C1 tokens.
2. At time T = 5 days, the Borrower borrows 50 C2 tokens.
3. At time T = 1 year, the Depositor decides to withdraw 10 C1 tokens.

In the current implementation, the amount of underlying tokens that the Depositor would receive is 10.018 C1 tokens. If instead the interest for C1 is accrued at the beginning of the withdrawal, then the Depositor would instead receive 10.611 C1 tokens. The Depositor loses out approximately 5.6% of the C1 tokens due to the missing interest accrual. However, we want to emphasize again that this assumes that there is no borrow, repay, or liquidation activity for C1 for the entire duration of a year, which is likely untrue if users are actively using C1 on the protocol.

Recommendation If the missing accrued interest updates are unintended behavior, then the developers should ensure that the accrued interest of each money market asset used as collateral is updated before any calls to `_exchangeRate()` and `_getValueOfCollateral()`.

Developer Response The developers stated that the current implementation is the intended behavior:

> This is how every lending protocol in DeFi works (check Compound's implementation as an example). Accruing interest on every market for every action would dramatically spike gas costs for the user and make the protocol unusable by hitting the max gas per block limit. This is also unnecessary because the amount of interest accrued is typically so small that it would not make a meaningful difference to the user's liquidity profile. Accrual is done before all actions that would cause a state change of that asset balance, so the user never loses funds they are entitled to when withdrawing. Furthermore, the protocol is not exposed to any risk because interest is always positive, and accruing a user's interest will only increase their balance. We do not believe this is an issue and in practice has no noticeable effect.

4.1.11 V-PRI3-VUL-011: Check for USP underlying does not check chain ID

Severity	Warning	**Commit**	ea69944
Type	Data Validation	**Status**	Fixed
File(s)	`PrimeOracle.sol`		
Location(s)	getBorrowAssetExchangeRate()		

The `getBorrowAssetExchangeRate()` has a special case for USP underlying. However, the check for this case only compares the underlying address against the USP address. In other methods such `getUnderlyingPriceBorrow()`, the special case for USP underlying also compares the underlying chain ID with the USP chain ID.

Impact It may be possible for an `underlying` address on one chain to collide with the USP address on another chain, in which case the wrong result will be returned.

Recommendation Check the chain ID of USP in `getBorrowAssetExchangeRate()`.

4.1.12 V-PRI3-VUL-012: Extra conversion to ptokens in in completeWithdraw

Severity	Warning	Commit	ea69944
Type	Logic Error	Status	Fixed
File(s)	PTokenMessageHandler.sol		
Location(s)	completeWithdraw()		

A user can request a collateral withdrawal by calling the `RequestController.withdraw()` function. Given the amount of ptokens to withdraw, the `withdraw()` method will send a withdrawal request to the master chain, where it will be processed by `MasterInternals._withdrawAllowed()`. The master state will update its bookkeeping and send back both the approval status and the `actualWithdrawAmount` amount of underlying collateral to send to the recipient of the withdraw request. When the satellite receives the response, the response will be routed to the `completeWithdraw()` method of the PToken contract. In `PTokenMessageHandler.completeWithdraw()`, the `actualWithdrawAmount` from the master state will be stored in `params.withdrawAmount`. This is multiplied by the external exchange rate (which was stored on the master state) to calculate the `actualWithdrawAmount` of underlying tokens to transfer to the withdraw recipient. The `params.withdrawAmount` is in units of underlying, which is inconsistent with the `params.externalExchangeRate`, which is in terms of underlying per ptoken. Furthermore, the calculation of `actualWithdrawAmount` already has a multiplication with the `externalExchangeRate` (which is part of the `_exchangeRate()`).

```
pTokenCollateralBalances[pTokenChainId][withdrawer][pToken] -= pTokenWithdrawAmount;
(uint256 exchangeRate, uint256 normalizeFactor) = _exchangeRate(
    pToken,
    pTokenChainId
);
actualWithdrawAmount = pTokenWithdrawAmount * exchangeRate / normalizeFactor;
```

Snippet 4.17: Calculation of `actualWithdrawAmount` in `MasterInternals._withdrawAllowed()`. The `ptokenWithdrawAmount` is the amount of ptokens specified by the user; multiplying it by the exchange rate converts it into underlying collateral.

```
function completeWithdraw(
    IHelper.FBWithdraw memory params
) external payable virtual override onlyMid() {
    if (isFrozen) revert MarketIsFrozen(address(this));
    emit WithdrawApproved(
        params.user,
        address(this),
        params.withdrawAmount,
        true
    );
    uint256 actualWithdrawAmount = (params.withdrawAmount * params.
    externalExchangeRate) / 10**EXCHANGE_RATE_DECIMALS;
    _doTransferOut(params.user, underlying, actualWithdrawAmount);
}
```

Snippet 4.18: Implementation of `PTokenMessageHandler.completeWithdraw()`

Impact There is no impact in the current version, because it is assumed that `PTokenMessageHandler` will be used with non-rebasing tokens, which have an external exchange rate of 1:1. However, if this assumption is violated in the future, this may lead to wrong amounts of underlying tokens being transferred to users.

Recommendation Fix the inconsistencies.

Developer Response The developers noted that the correct behavior is to send the underlying amount from master, and they have changed the code in `completeWithdraw()`.

4.1.13 V-PRI3-VUL-013: Redundant comparison against 0

Severity	Warning	**Commit**	ea69944
Type	Logic Error	**Status**	Fixed
File(s)	DIAFeedGetter.sol		
Location(s)	getAssetPrice()		

In `getAssetPrice()`, the `answer` from the DIA oracle is an unsigned integer, so the expression `answer < 0` will always evaluate to `false`. This means that `return (0, 0)` is dead code.

```
(
    uint128 answer,
    /* uint128 timestampe */
) = feed.getValue(key);

if (answer < 0) return (0,0);
```

Snippet 4.19: Location of the comparison in `getAssetPrice()`

Recommendation Change the check to `answer <= 0` (if attempting to guard against invalid prices) or remove the check altogether.

4.1.14 V-PRI3-VUL-014: Contradictory comments on RequestController.withdraw

Severity	Warning	**Commit**	ea69944
Type	Maintainability	**Status**	Fixed
File(s)	`RequestController.sol`		
Location(s)	withdraw()		

A documentation comment on `RequestController.withdraw()` indicates that a deposit can call it with ptokens in exchange for underlying assets. However, the comment on `withdrawAmount` states that the amount is denominated in underlying collateral tokens. Furthermore, based on the way the variable is used when it is sent over to the master contract, it appears to actually be in terms of ptokens in practice. This comment could mislead the developers into introducing unit conversion bugs in the future.

```
1  /**
2    * @notice Depositor withdraws pTokens in exchange for a specified amount of
       underlying asset
3    * @param withdrawAmount The amount of underlying to withdraw from the protocol
4    * @param route Route through which to request to withdraw tokens
5    */
6    function withdraw(
7        address route,
8        uint256 withdrawAmount,
9        address pToken,
10       uint256 targetChainId
11   ) external override payable nonReentrant() {
```

Snippet 4.20: Comments on the `withdraw()` method

Impact The developers should resolve the contradictory comments and fix the intended behavior if needed.

Developer Response The developers noted that `withdrawAmount` is in terms of ptokens.

4.1.15 V-PRI3-VUL-015: Potentially missing special case for USP in getUnderlyingPrice

Severity	Warning	Commit	ea69944
Type	Logic Error	Status	Intended Behavior
File(s)	PrimeOracle.sol		
Location(s)	getUnderlyingPrice()		

In `getUnderlyingPriceBorrow()` and `getBorrowAssetExchangeRate()`, there are special cases for the USP token. For example, `getUnderlyingPriceBorrow()` will directly return 1 as the price for the USP token instead of consulting an oracle. However, the `getUnderlyingPrice()` is missing a similar special case for USP.

```
function getUnderlyingPrice(
    uint256 chainId,
    address collateralMarketUnderlying
) external view override returns (uint256, uint8) {
    return _getAssetPrice(chainId, collateralMarketUnderlying);
}

function getUnderlyingPriceBorrow(
    uint256 chainId,
    address loanMarketUnderlying
) external view override returns (uint256, uint8) {
    if (uspAddress == address(0)) revert UspAddressZero();

    if (loanMarketUnderlying == uspAddress && chainId == block.chainid) {
        uint8 uspDecimals = ERC20(uspAddress).decimals();
        return (10**uspDecimals, uspDecimals);
    } else {
        return _getAssetPrice(chainId, loanMarketUnderlying);
    }
}
```

Snippet 4.21: Implementation of `getUnderlyingPrice()` and `getUnderlyingPriceBorrow()`

Recommendation The developer should clarify the intended behavior and role of the USP token in this contract.

Developer Response The developers noted that this is intended:

> getUnderlyingPrice() is used internally in _getValueofCollateral() and _syncAsset-Value(), which both only call getUnderlyingPrice() on PTokens, so a special case for USP would be dead code as it would never execute as USP can't be used as collateral

4.1.16 V-PRI3-VUL-016: _syncUserRewards always applies to msg.sender

Severity	Warning	Commit	ea69944
Type	Maintainability	Status	Fixed
File(s)	StakingPoolInternals.sol		
Location(s)	_syncUserRewards()		

The `StakingPool` allows users to stake assets and collect rewards for them. At every user interaction, the pool contract calls an internal function `_syncUserRewards()`, which calculates the accrued rewards for the `msg.sender` since the last update for the `msg.sender`. In the current version,

```
function _syncUserReward(address rewardsToken) internal {
    uint256 rewardsIndexDiff = (rewardMarkets[rewardsToken].marketIndex - userRewards
    [msg.sender][rewardsToken].rewardsIndex);
    uint256 rewardsAccumulated = rewardsIndexDiff * userBalances[msg.sender] / 10**
    STAKING_REWARDS_DECIMALS;
    userRewards[msg.sender][rewardsToken].accumulatedRewards += rewardsAccumulated;
    userRewards[msg.sender][rewardsToken].rewardsIndex = rewardMarkets[rewardsToken].
    marketIndex;
}
```

Snippet 4.22: Function `_syncUserReward` in StakingPoolInternals.sol

the function is called whenever a user stakes, unstakes, or claims rewards. `_syncRewardsUser()` assumes that the `msg.sender` is always the user for whom the rewards should be updated.

Impact The issue described above currently has no impact; however, if the developers intend to add additional functionality, such as the ability to stake on behalf of another user, then they may forget to that `_syncUserReward()` only applies to the `msg.sender`.

Recommendation The `_syncUserRewards()` should explicitly take the address of the user for whom the rewards should be updated for.

4.1.17 V-PRI3-VUL-017: MAX_SIZE can be defined in terms of other constants

Severity	Warning	**Commit**	ea69944
Type	Maintainability	**Status**	Acknowledged
File(s)	`ECC.sol`		
Location(s)	See description		

The `ECC` contract has three constants `MAX_SIZE`, `METADATA_SIZE`, and `USABLE_SIZE` corresponding to the total size of a message, the size of the metadata part, and the maximum size of the payload part, respectively. However, they are all defined as hardcoded constants, even though there is an implicit assumption that `MAX_SIZE == METADATA_SIZE + USABLE_SIZE`.

Impact If the developers change either `METADATA_SIZE` or `USABLE_SIZE`, they may forget to update `MAX_SIZE`. This may result in serious errors, as the `MAX_SIZE` determines the storage layout of the messages stored in the contract.

Recommendation Define `MAX_SIZE` as:

```
1 uint256 internal constant MAX_SIZE = METADATA_SIZE + USABLE_SIZE
```

Developer Response The developers noted:

> The ECC contract currently on Mainnet cannot be upgraded, so to maintain consistency between live contracts and our repository we won't be making these changes locally. However in future versions, or in a case where we need to redeploy this contract, we will make these updates to improve readability/maintainability.

4.1.18 V-PRI3-VUL-018: setAssetKey does not check whether feed exists

Severity	Warning	**Commit**	ea69944
Type	Data Validation	**Status**	Acknowledged
File(s)	`DIAFeedGetter.sol`		
Location(s)	setAssetKey()		

The `setAssetKey()` function can be called by an admin to set the key that will be looked up for a particular DIA feed. However, it does not validate whether the corresponding feed actually exists.

```
function setAssetKey(
    uint256 chainId,
    address asset,
    string memory key
) external onlyAdmin() {
    assetKeys[chainId][asset] = key;
    emit AssetKeyUpdated(chainId, asset, key);
}
```

Snippet 4.23: Implementation of `setAssetKey()`

Impact An admin may make the mistake of setting an asset key for a feed that does not exist, and there is nothing preventing this mistake from happening.

Recommendation Require that `assetFeeds[chainId][asset]` is nonzero.

Developer Response The developers noted that this function is meant to be called during deployment, and they do not want to enforce a strict order between calls to `setAssetFeed()` (used to set the `assetFeeds[chainId][asset]` entry) and `setAssetKey()`.

Furthermore, they stated that:

> For DIA feeds both the feed and key need to be set for them to properly work, so adding just a check on setAssetKey doesn't make sense without also having one on setAssetFeed. The problem with this is at runtime, there is no guarantee the order in which the admin will call these setters, so there's not a maintainable way to add such sanity checks.

4.1.19 V-PRI3-VUL-019: Price bounds are magic constants

Severity	Warning	**Commit**	ea69944
Type	Maintainability	**Status**	Fixed
File(s)	`TreasuryAdmin.sol`		
Location(s)	supportLoanAsset(), modifyTradeAsset()		

The `supportLoanAsset()` and `modifyTradeAsset()` methods both provide a `_mintPrice` argument and a `_burnPrice` argument, which are fixed point integers with `FACTOR_DECIMALS` (a constant equal to 8) decimals. However, the validation in both methods use magic constants `1e8` and `5e6`. These magic constants are equal to `10**FACTOR_DECIMALS` and `5 * 10**(FACTOR_DECIMALS - 2)`, respectively.

```
/* Min: 1e8 */
/* Max: 105e6 */
if (_mintPrice - 1e8 > 5e6) revert ParamOutOfBounds();
if (_burnPrice - 1e8 > 5e6) revert ParamOutOfBounds();
```

Snippet 4.24: The range checks for `_mintPrice` and `_burnPrice`.

Impact If the `FACTOR_DECIMALS` is modified, and the developers forget to update these magic constants, then the bounds checks will be incorrect.

Recommendation Replace the magic constants with appropriately named constant variables.

4.1.20 V-PRI3-VUL-020: Unused mappings in PrimeOracleStorage

Severity	Info	**Commit**	ea69944
Type	Maintainability	**Status**	Fixed
File(s)	`PrimeOracleStorage.sol`		
Location(s)	See description		

The storage variables `exchangeRatePrimaryFeeds` and `exchangeRateSecondaryFeeds` of the `PrimeOracleStorage` abstract contract are not used anywhere. Since the `PrimeOracle` is not based on an upgradable proxy, these storage variables can be safely removed.

4.1.21 V-PRI3-VUL-021: Confusing naming in supportSatelliteLoanMarket

Severity	Info	**Commit**	ea69944
Type	Maintainability	**Status**	Fixed
File(s)	`MasterAdmin.sol`		
Location(s)	supportSatelliteLoanMarket()		

The auditors found the naming of the parameters of the `MasterAdmin.supportSatelliteLoanMarket()` method to be very confusing. We recommend that the developers clarify the naming and add documentation comments explaining what the parameters are and how the method is meant to be used.

4.1.22 V-PRI3-VUL-022: Inconsistent naming in TreasuryAdmin methods

Severity	Info	**Commit**	ea69944
Type	Maintainability	**Status**	Fixed
File(s)	`TreasuryAdmin.sol`		
Location(s)	See description		

The `TreasuryAdmin` contract contains three methods named `supportLoanAsset()`, `removeTradeAssetToLoanAsset()`, and `modifyTradeAsset()`, which operate over the same data structures. The name of `modifyTradeAsset()` is inconsistent with the other two names.

Developer Response The developers renamed `supportLoanAsset` to `supportTradeAsset` and `removeTradeAssetToLoanAsset` to `removeTradeAsset`.

