Veridise. **Auditing Report**

Hardening Blockchain Security with Formal Methods

FOR

Prime Protocol

Veridise Inc.
April 11, 2023

► **Prepared For:**

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

► **Prepared By:**

Bryan Tan
Ajinkya Rajput

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

► **Version History:**

Apr. 11, 2023	V1
Apr. 5, 2023	Initial Draft

Contents

Executive Summary 1

From Mar. 9, 2023 to Mar. 31, 2023, Prime engaged Veridise to review the security of their Prime Protocol. The review covered an extension to the Prime Protocol that implements a new money markets feature that allows users to borrow from collateral token deposits*. Compared to the previous version, which Veridise has audited previously†, the code has been significantly revised to accommodate the new feature. Veridise conducted the assessment over 6 person-weeks, with 2 engineers reviewing code over 3 weeks on Git commit `b1ee399`. 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 developers provided the source code of the Prime Protocol contracts for review. To facilitate the Veridise auditors' understanding of the code, the Prime developers shared a whitepaper describing the high-level functionality of the protocol as well as internal documentation of some of the protocol's implementation details. The source code also contained some documentation in the form of READMEs and documentation comments on functions and storage variables.

The source code contained a test suite, which the Veridise auditors noted only provides partial coverage, particularly in the newly added code as we note in the suggestions below. Several files in the source code also indicate that the developers use linting and static analysis tools such as Solhint and Slither, respectively.

Summary of issues detected. The audit uncovered 29 issues, the majority of which are related to bugs in the implementation. Among all issues, 6 are assessed to be of high or critical severity by the Veridise auditors, including an issue where users may be unable to withdraw collateral after an admin lists a new loan market that uses that collateral (V-PRI2-VUL-006); multiple issues where numerical units are inconsistent (V-PRI2-VUL-002, V-PRI2-VUL-003, V-PRI2-VUL-004) in a way that disrupts the intended behavior of the protocol; as well as another issue where users will be unable to repay loans that use native currency as collateral (V-PRI2-VUL-005). The Veridise auditors also identified several medium-severity issues, including rounding errors leading to reverts (V-PRI2-VUL-007) and potential locked funds due to missing data validation (V-PRI2-VUL-009). Additionally, the auditors reported a number of minor issues, including 7 low-severity issues and 13 warnings.

The Prime developers fixed all of the issues reported by the auditors.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the protocol's long term safety.

* The money markets feature is described in the blog post: `https://medium.com/prime-protocol/introducing-money-markets-on-prime-protocol-dcabf6472b93`

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

Test coverage. Several issues relate to missing data validation and/or bugs in the core calculations, such as V-PRI2-VUL-005, V-PRI2-VUL-008, and V-PRI2-VUL-009. We believe these issues could have been caught while testing realistic user behaviors, such as:

- Cases where native currency is used as collateral for a loan.
- Cases where a PToken with an external exchange rate greater than 1 is used as collateral and/or loaned out as a money market asset.
- Cases where users mistakenly provide bad or invalid PToken or lendable asset addresses as parameters.
- Cases where users interact with multiple assets that are each configured differently (e.g., borrowing a first-party loan asset by using a rebasing token as collateral).

PTokens with an external exchange rate. Several of the high severity issues discovered by the Veridise auditors involve the cases where the collateral-per-PToken exchange rate is not equal to one. As such, we recommend that the developers thoroughly test and debug such PTokens end-to-end before they attempt to deploy such PTokens.

Maintainability. The Veridise auditors observed that some of the issues discovered in the audit may have been a consequence of maintainability issues. In particular, V-PRI2-VUL-002 may have been caused by ambiguous variables names such as `actualWithdrawAmount` causing confusion during development. We recommend naming ambiguous variables and field names such as `totalSupply` and `totalSupplied` to clearer, unambiguous names such as `totalPtokenSupply` and `totalCollateralSupply`.

Additionally, the auditors encountered multiple locations in the code where the developers make implicit assumptions about invariants that should hold (e.g., see V-PRI2-VUL-013 and developer responses in V-PRI2-VUL-002 and V-PRI2-VUL-003). To reduce the possibility that a future change violates these assumptions and introduce bugs, we recommend that the developers clearly document such assumptions in the code and check that the invariants hold during testing.

Furthermore, to prevent unit conversion or fixed point arithmetic issues such as V-PRI2-VUL-001 from occurring in the future, we recommend that the developers document the units and precision of each parameter and state variable, and that they should also insert comments where unit and/or precision conversions may occur (V-PRI2-VUL-029).

Finally, the Veridise auditors observed that the new changes have introduced higher degrees of branching and special cases, which results in higher code complexity overall compared to the previous version. This has made the code more error-prone (e.g., V-PRI2-VUL-006) and susceptible to mistakes when performing upgrades. If possible, we recommend that the Prime developers simplify their protocol by removing as many special cases as possible. For example, they could remove the native currency underlying special case in the PToken contracts and instead use a wrapped native currency token as underlying.

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.

Project Dashboard 2

Table 2.1: Application Summary.

Name	Version	Type	Platform
Prime Protocol	`b1ee399`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 9 - Mar. 31, 2023	Manual & Tools	2	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	6	6
Medium-Severity Issues	3	3
Low-Severity Issues	7	7
Warning-Severity Issues	13	13
Informational-Severity Issues	0	0
TOTAL	29	29

Table 2.4: Category Breakdown.

Name	Number
Logic Error	10
Maintainability	8
Data Validation	4
Locked Funds	2
Denial of Service	2
Missing/Incorrect Events	2
Usability Issue	1

Audit Goals and Scope 3

3.1 Audit Goals

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

- Does the money market functionality work as intended?
- Are all user-initiated flows such as deposits, borrows, etc. fault tolerant?
- Are loan interest rates calculated correctly?
- Are gas fees forwarded to the correct contracts and to the correct refund addresses?
- Are all unit conversions implemented correctly and with the correct precision?
- Do all admin functions validate their arguments, and what impact does misconfiguration have?
- It is possible for user funds to be locked in the protocol, including accidentally or by an attacker?
- Can an attacker abuse missing validation to cause financial damage to the protocol and other users?
- Can an attacker withdraw more funds than they are entitled to?
- What possible strategies could attackers employ to forcefully trigger liquidations?

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 the following files:

- The files that both (1) were in scope in the previous audit; and (2) are contained in one of the following folders:
  - `ecc`
  - `interfaces`
  - `master`

  - `middleLayer`
  - `satellite`
  - `util`
- The following new source files written by the developers:
  - `satellite/pToken/extensions/*`
  - `satellite/pToken/implementations/*`
  - `satellite/requestController/*`
  - `master/irm/implementations/doubleLinear/*`

Several files/components were also explicitly excluded from the scope, including:

- `ECC.sol`
- `Axelar.sol`
- `Wormhole.sol`
- `PrimeOracle.sol`
- `ChainlinkFeedGetter.sol`
- `Staking.sol`
- `MultiStaticCall.sol`
- `Treasury.sol`

During the audit, the Veridise auditors referred to the excluded files but assumed that they have been implemented correctly.

Methodology. Veridise auditors reviewed the report of the previous audit conducted by Veridise for Prime, inspected the provided tests, and read the documentation provided by the Prime developers. 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 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:

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:

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

Vulnerability Report

4

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-PRI2-VUL-001	protocolIncentive divided by wrong precision fa. .	High	Fixed
V-PRI2-VUL-002	Liquidity check compares different units in _w. . .	High	Fixed
V-PRI2-VUL-003	Satellite loan market exchange rate calculation. . .	High	Fixed
V-PRI2-VUL-004	_getValueOfCollateral multiplies wrong units	High	Fixed
V-PRI2-VUL-005	Native currency collateral repayment always rev. .	High	Fixed
V-PRI2-VUL-006	Loan market totalSupplied inconsistency after c. . .	High	Fixed
V-PRI2-VUL-007	Rounding error may cause Aave PToken withdraw	Medium	Fixed
V-PRI2-VUL-008	RequestController.deposit does not forward msg.. .	Medium	Fixed
V-PRI2-VUL-009	Liquidating loan asset of zero locks native cur. . .	Medium	Fixed
V-PRI2-VUL-010	Potential rounding error causes revert in query. . .	Low	Fixed
V-PRI2-VUL-011	changeProtocolIncentive does not validate bounds	Low	Fixed
V-PRI2-VUL-012	withdrawReserves refunds gas to wrong account	Low	Fixed
V-PRI2-VUL-013	PToken does not validate decimals of underlying	Low	Fixed
V-PRI2-VUL-014	Minor rounding error when calculating liquidati. .	Low	Fixed
V-PRI2-VUL-015	supportMarket can be called on a previously lis. . .	Low	Fixed
V-PRI2-VUL-016	Associated loan market not validated before use	Low	Fixed
V-PRI2-VUL-017	Same hash hardcoded in two locations	Warning	Fixed
V-PRI2-VUL-018	Duplicated logic in PToken deposit, depositBehalf	Warning	Fixed
V-PRI2-VUL-019	Missing events in RewardControllerAdmin	Warning	Fixed
V-PRI2-VUL-020	Duplicated initialization logic in PToken imple. . .	Warning	Fixed
V-PRI2-VUL-021	Buffer overflow in MiddleLayer._mreceive	Warning	Acknowledged
V-PRI2-VUL-022	Potentially incorrect cast in unlockLiquidation. . .	Warning	Fixed
V-PRI2-VUL-023	Unfairness while withdrawing collateral in low-. . .	Warning	Intended Behavior
V-PRI2-VUL-024	Inconsistent comments in DoubleLinearIRMStorag	Warning	Fixed
V-PRI2-VUL-025	collateralBalances is confusingly named	Warning	Fixed
V-PRI2-VUL-026	Liquidation response can forward msg.value twice	Warning	Acknowledged
V-PRI2-VUL-027	Implicit interface is shared by PToken and Loan. . .	Warning	Fixed
V-PRI2-VUL-028	Missing events on interest accrual	Warning	Fixed
V-PRI2-VUL-029	Consider documenting units in calculations	Warning	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-PRI2-VUL-001: protocolIncentive divided by wrong precision factor

Severity	High	**Commit**	b1ee399
Type	Logic Error	**Status**	Fixed
File(s)	`[...]/ASVTokenV3RewardsController.sol`		
Location(s)	_withdrawRewards()		

In `ASVTokenV3RewardsController._withdrawRewards()`, a percentage of the user's claimed rewards is deducted as a fee. This computation is carried out by multiplying the `claimedRewards` with the `protocolIncentive` , and then dividing by a precision constant to ensure that the result has the correct precision. However, the precision factor used is `FACTOR_DECIMALS` (an exponent) rather than `10**FACTOR_DECIMALS` (the actual precision factor).

```
uint256 protocolShare = claimedRewards * protocolIncentive / FACTOR_DECIMALS;
uint256 adjustedClaimedRewards = claimedRewards - protocolShare;
```

Snippet 4.1: The location in `_withdrawRewards()` where the user's claimed rewards are adjusted.

Impact Because `FACTOR_DECIMALS` is significantly smaller than `10**FACTOR_DECIMALS`, the `protocolShare` will be much larger than it should be. This will cause the subtraction overflow in `claimedRewards - protocolShare`, leading to a revert when depositing, withdrawing, or liquidating. If no revert occurs, the adjusted claimed rewards will be significantly reduced compared to the actual intended amount.

Recommendation Change `FACTOR_DECIMALS` to `10**FACTOR_DECIMALS`.

Developer Response The developers indicated that they will be removing the rewards controller logic entirely, as they do not intend to use it in the future.

4.1.2 V-PRI2-VUL-002: Liquidity check compares different units in _withdrawAllowed

Severity	High	**Commit**	b1ee399
Type	Logic Error	**Status**	Fixed
File(s)	master/MasterInternals.sol		
Location(s)	_withdrawAllowed()		

The `MasterInternals._withdrawAllowed()` method determines whether a collateral withdrawal initiated on a satellite chain should be approved. Part of the approval check is to determine whether the protocol has sufficient liquidity to cover the withdrawal; if the protocol lacks liquidity, then the `_withdrawAllowed()` method will revert.

In the check, the parameter `pTokenWithdrawAmount` is given in terms of the amount of PTokens to withdraw. This value is compared against the `totalSupplied - totalBorrows` , which are underlying collateral amounts and not PToken amounts.

```
if (totalSupplied != type(uint256).max) {
    uint256 totalBorrows = loanMarkets[loanMarket.loanAsset][loanMarket.chainId].
    totalBorrows;
    if (totalBorrows > totalSupplied || (totalSupplied - totalBorrows) <
    pTokenWithdrawAmount) {
        revert InsufficientLiquidity();
    }

    loanMarkets[loanMarket.loanAsset][loanMarket.chainId].totalSupplied -=
    actualWithdrawAmount;
}
```

Snippet 4.2: The snippet in `_withdrawAllowed()` that checks whether there is sufficient liquidity for a withdrawal.

Impact If the collateral-per-PToken exchange rate is not 1:1, a withdrawal may not be approved even if there is sufficient liquidity to cover the withdrawal, or a withdrawal may be approved even if there is insufficient liquidity.

Recommendation The actual amount of collateral that is withdrawn is already computed in the local variable `actualWithdrawAmount`, which is computed as the `pTokenWithdrawAmount` multiplied by the collateral-per-PToken exchange rate. The liquidity check should compare against `actualWithdrawAmount`, not `pTokenWithdrawAmount`.

```
actualWithdrawAmount = pTokenWithdrawAmount * exchangeRate / normalizeFactor;
```

Snippet 4.3: The line that computes `actualWithdrawAmount`

Developer Response The developers noted the following:

> Since the PToken exchange rate is strictly increasing (we can't have negative interest), and `actualWithdrawAmount = pTokenWithdrawAmount * exchangeRate / normalizeFactor` , then the boolean "(totalSupplied - totalBorrows) < pTokenWithdrawAmount" will never reject a withdrawal when there is sufficient liquidity. This means funds can't get locked as a result. I think this more accurately describes the impact:
>
> "If the collateral-per-PToken exchange rate is not 1:1, a withdrawal may be approved even if there is insufficient liquidity. This would lead to the withdrawal failing on satellite, and the withdrawal message would need to be re-sent when there is more liquidity in the pool"
>
> This issue does not lead to any funds being locked incorrectly, or the incorrect distribution of any funds.

4.1.3 V-PRI2-VUL-003: Satellite loan market exchange rate calculation uses wrong units

Severity	High	**Commit**	b1ee399
Type	Logic Error	**Status**	Fixed
File(s)	`master/MasterInternals.sol`		
Location(s)	_exchangeRate()		

The `_exchangeRate()` internal function is used to calculate the underlying-collateral-per-PToken exchange rate (`exchangeRate`) for a given PToken. This calculation is performed by (1) multiplying the total quantity of PToken available for borrowing by (2) the PToken's collateral-per-PToken exchange rate, and (3) dividing by the total quantity of PToken in circulation.

Specifically, when calculating the `exchangeRate`, the involved variables and units are:

1. `totalSupply` is in terms of PTokens
2. `totalSupplied` is in terms of:
   - PTokens if the `chainId` is zero
   - underlying collateral if the `chainId` is nonzero
3. `externalExchangeRate` is in terms of collateral-per-PToken

For PTokens without an associated loan market (i.e., `chainId` is zero), the `exchangeRate` simplifies to term (2). However, the calculation does not make sense for PTokens with an associated loan market (i.e., `chainId` is nonzero), because it multiplies collateral by the collateral-per-PToken exchange rate.

Impact If the exchange rate between collateral and PToken is not 1:1, this will result in the calculated `exchangeRate` being incorrect. Because the `_exchangeRate()` method is used in several important methods like `_deposit()`, `_withdrawAllowed()`, and `_getValueOfCollateral()`, a mistake in `_exchangeRate()` could lead to issues in core parts of the protocol.

Recommendation The developers should clarify the behavior of `_exchangeRate()` when a PToken has an associated loan market and check that the units in the calculations are correct.

Developer Response The developers have corrected the units by changing the calculation to the following:

```
uint256 totalSupply = markets[pTokenChainId][pToken].totalSupply;
uint256 totalSupplied = totalSupply * markets[pTokenChainId][pToken].
    externalExchangeRate;

{
    LoanMarketMetadata memory targetMarket = mappedLoanAssets[pTokenChainId][pToken];

    if (targetMarket.chainId != 0) {
        totalSupplied = loanMarkets[targetMarket.loanAsset][targetMarket.chainId].
    totalSupplied * normalizeFactor;
    }
}
```



```
function _exchangeRate(
    address pToken,
    uint256 pTokenChainId
) public virtual override view returns (uint256 /* exchangeRate */, uint256 /* normalizeFactor */) {
    uint256 normalizeFactor = 10 ** EXCHANGE_RATE_DECIMALS;

    uint256 totalSupply = markets[pTokenChainId][pToken].totalSupply;
    uint256 totalSupplied = totalSupply;

    {
        LoanMarketMetadata memory targetMarket = mappedLoanAssets[pTokenChainId][pToken];

        if (targetMarket.chainId != 0) {
            totalSupplied = loanMarkets[targetMarket.loanAsset][targetMarket.chainId].totalSupplied;
        }
    }

    if (totalSupplied == 0 ||  totalSupplied == type(uint256).max) return (normalizeFactor, normalizeFactor);

    uint256 numerator = totalSupplied * markets[pTokenChainId][pToken].externalExchangeRate;
    uint256 denominator = totalSupply;
    uint256 exchangeRate = numerator / denominator;

    return (exchangeRate, normalizeFactor);
}
```

Snippet 4.4: Implementation of `_exchangeRate()`

```

if (totalSupplied == 0 ||  totalSupplied == type(uint256).max) return (normalizeFactor, normalizeFactor);

uint256 exchangeRate = totalSupplied / totalSupply;
```

They noted that the `markets[pTokenChainId][pToken].externalExchangeRate` is assumed to be greater than one (in the same scale as `normalizeFactor`).

4.1.4 V-PRI2-VUL-004: _getValueOfCollateral multiplies wrong units

Severity	High	**Commit**	b1ee399
Type	Logic Error	**Status**	Fixed
File(s)	master/MasterInternals.sol		
Location(s)	_getValueOfCollateral()		

The method `_getValueOfCollateral()` computes the US$ value of a user's deposit in one collateral market (with optional parameters to simulate the effect of a borrow or a withdraw). However, when computing this value, one of the multiplications does not multiply quantities of the correct units. The function first calculates a factor `tokensToDenom` , which represents the US$ value of each underlying collateral token after adjusting for initial/maintenance collateral ratios:

```
tokensToDenom = exchangeRate * collateralFactor * oraclePrice / 10**factorDecimals /
    10**oracleDecimals;
```

The units of the variables are:

- `exchangeRate` is in terms of underlying tokens per PToken
- `collateralFactor` is a percentage term (i.e., unitless)
- `oraclePrice` is the US$ price per underlying token

Thus, the units of `tokensToDenom` is US$ price per PToken, as a fixed point number with the same precision as that of `exchangeRate` .

Then the underlying collateral balance of the user is retrieved from the function `_collateralBalanceStored()` and stored in variable `collBall` . The implementation of `_collateralBalanceStored()` multiplies the user's balance of the PToken by the underlying-collateral-per-PToken exchange rate. Therefore, inside of `_getValueOfCollateral()`, `collBall` is in terms of number of underlying collateral tokens.

Finally, the calculation of `collateralValue` multiplies `tokenToDenom` by `collBal` . Here, the US$ value per PToken is multiplied by amount of underlying collateral tokens. Thus, the final quantity is in US$ times collateral per PToken, not in the expected units of US$.

Impact If the collateral-per-PToken exchange rate is not 1:1, the US$ value of collateral of a user will be calculated incorrectly. This may cause the protocol to over- or under-value the collateral of the user.

Recommendation Because the exchange rate is already accounted for in `_collateralBalanceStored()`, the `exchangeRate` factor in `tokensToDenom` is unnecessary. The calculation for `tokensToDenom` should be changed to:

```
tokensToDenom = collateralFactor * oraclePrice / 10**factorDecimals / 10**
    oracleDecimals;
```

This will change the units of `tokensToDenom` to US$ value per collateral token, which will correct the unit issue with `collateralValue`. The developers may also need to include an additional precision factor to avoid rounding issues.

Note that if the `exchangeRate` factor is removed from `tokensToDenom`, then the calculation of `withdrawEffect` will need to incorporate the `exchangeRate` factor:

```
- withdrawEffect = (tokensToDenom * withdrawAmount) / pTokenDecimals; /* normalize */
+ withdrawEffect = (exchangeRate * tokensToDenom * withdrawAmount) / pTokenDecimals;
    /* normalize */
```

Developer Response The developers changed the calculation of `collateralValue` to:

```
uint256 collateralValue;
{
    // Note: We don't use '_collateralBalanceStored()' here since the exchangeRate
    has already been applied in 'tokensToDenom'
    uint256 collBal = collateralBalances[pTokenChainId][user][pToken];

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

This should have the same effect as our recommendation above, as the exchange rate will not be multiplied twice.

```
function _getValueOfCollateral(
    address user,
    address pToken,
    uint256 pTokenChainId,
    uint256 borrowAmount,
    uint256 withdrawAmount
) private view returns (uint256 /* collateralValue */, uint256 /* withdrawEffect */) {
    uint256 tokensToDenom;
    {
        uint256 collateralFactor; uint256 factorDecimals; uint256 oraclePrice; uint256 oracleDecimals;

        {
            address pTokenUnderlying = markets[pTokenChainId][pToken].underlying;

            if (borrowAmount != 0 || withdrawAmount != 0) { /* Borrow/Withdraw */
                (collateralFactor, factorDecimals) = collateralRatioModel.getCollateralFactor(pTokenChainId, pTokenUnderlying);
            } else { /* Liquidate */
                (collateralFactor, factorDecimals) = collateralRatioModel.getMaintenanceCollateralFactor(pTokenChainId, pTokenUnderlying);
            }
            //usd per collateral
            (oraclePrice, oracleDecimals) = oracle.getUnderlyingPrice(pTokenChainId, pTokenUnderlying);

            if (oraclePrice == 0) revert InvalidPrice();
        }

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

    uint256 pTokenDecimals = 10**markets[pTokenChainId][pToken].decimals;
    uint256 collateralValue;
    {                           //in collateral
        uint256 collBal = _collateralBalanceStored(user, pToken, pTokenChainId);
                        // usd/ptoken      no.collater
        collateralValue = tokensToDenom * collBal / pTokenDecimals;
    }
```

Snippet 4.5: Relevant snippet in `_getValueOfCollateral()`

```
uint256 collateralBalance = collateralBalances[pTokenChainId][account][pToken];

/* If collateralBalance = 0 then collateralIndex is likely also 0.
* Rather than failing the calculation with a division by 0, we immediately return 0
    in this case.
*/
if (collateralBalance == 0) {
    return 0;
}

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

return collateralBalance * exchangeRate / normalizeFactor;
```

Snippet 4.6: Snippet from `_collateralBalanceStored()` that calculates the user's collateral balance.

4.1.5 V-PRI2-VUL-005: Native currency collateral repayment always reverts

Severity	High	**Commit**	b1ee399
Type	Logic Error	**Status**	Fixed
File(s)	`satellite/pToken/PTokenBase.sol`		
Location(s)	processRepay()		

A user may repay a loan by invoking `RequestController.repayBorrow()` on a PToken or LoanAsset that they would like to repay. This method will invoke `RequestControllerInternals._sendRepay()`, which will first call the `PToken.processRepay()` or `LoanAsset.processRepay()` method before sending a message to the master state to update the bookkeeping.

If the loan is for a PToken, the `processRepay()` method will calls the `_doTransferFrom()` helper function from `utils/SafeTransfers.sol` to move the funds from the `repayer` address to the PToken contract. However, `_doTransferFrom()` does not implement the native currency case (where `underlying` is the zero address) and will revert.

```
function processRepay(
    address repayer,
    uint256 repayAmount
) external /* override */ onlyRequestController() {
    if (repayAmount == 0) revert AmountIsZero();

    _doTransferFrom(repayer, address(this), underlying, repayAmount);
}
```

Snippet 4.7: Implementation of `PTokenBase.processRepay()`

```
function _doTransferFrom(
    address from,
    address to,
    address underlying,
    uint256 amount
) internal virtual returns (uint256) {
    if (from == address(this)) {
        revert("Use _doTransferOut()");
    }
    if (underlying == address(0)) {
        revert("Requires manual impl");
    }
```

Snippet 4.8: Location in `_doTransferFrom()` where the revert occurs.

Impact A user will be unable to repay a loan that uses native currency as underlying collateral.

Recommendation The developers should implement a mechanism that handles the native currency case.

Developer Response The developers changed the call to `_doTransferFrom()` to `_doTransferIn()`, which checks `msg.value` to ensure that the caller has supplied a sufficient amount of native currency. They also added logic in several of the `RequestController` methods to correctly handle PTokens that use native currency as underlying.

4.1.6 V-PRI2-VUL-006: Loan market totalSupplied inconsistency after call to supportSatelliteLoanMarket()

Severity	High	**Commit**	b1ee399
Type	Locked Funds	**Status**	Fixed
File(s)	`master/MasterAdmin.sol`, `master/MasterInternals.sol`		
Location(s)	supportSatelliteLoanMarket(), _deposit()		

When users deposit funds into a PToken contract in a satellite chain, the master contract updates the protocol's bookkeeping in the `MasterInternals._deposit()` method. If the PToken is associated with a loan market, then the amount of underlying collateral for that loan market will be updated in the `loanMarkets[][].totalSupplied` field. A PToken does not have an associated

```
function _deposit(
    address depositor,
    address pToken,
    uint256 pTokenChainId,
    CollateralMarketType marketType,
    uint256 externalExchangeRate,
    uint256 exchangeRateTimestamp,
    uint256 depositAmount
) external payable virtual override {
      // ...
      {
          LoanMarketMetadata memory targetMarket = mappedLoanAssets[pTokenChainId][
    pToken];
          if (loanMarkets[targetMarket.loanAsset][targetMarket.chainId].totalSupplied
     != type(uint256).max) {
              loanMarkets[targetMarket.loanAsset][targetMarket.chainId].totalSupplied
     += depositAmount;
          }
      }

      markets[pTokenChainId][pToken].totalSupply += actualDepositAmount;
      // ...
```

Snippet 4.9: Relevant lines in `MasterInternals._deposit()`

loan market by default. In order to associate a loan market with a PToken, a protocol admin must invoke `MasterAdmins.supportSatelliteLoanMarket()` with the loan asset (`mappedLoanAsset`) and the PToken (`satelliteLoanAsset`). However, note that neither `supportSatelliteLoanMarket()` nor `supportLoanMarket()` updates the `totalSupplied` field of the loan market for the loan asset, despite the PToken's underlying token now serving as collateral for the loan asset. Thus, if some deposits occur before the admin invokes `supportSatelliteLoanMarket()` to associate the PToken with a loan asset, then those collateral amounts deposited will not count towards the collateral amounts available for the associated loan asset.

Impact The undercounting has at least two effects, including preventing withdrawals and preventing borrows.

When a user requests a withdraw, the `_withdrawAllowed()` method checks that there is sufficient liquidity to ensure that all loans are backed by collateral. If the protocol has called


```
function supportSatelliteLoanMarket(
    address mappedLoanAsset,
    uint256 mappedLoanAssetChainId,
    address satelliteLoanAsset,
    uint256 satelliteLoanAssetChainId
) external override onlyAdmin() {
    if (!loanMarkets[mappedLoanAsset][mappedLoanAssetChainId].isListed) revert
    LoanMarketIsListed(false);

    LoanMarketMetadata memory _metadata;
    _metadata.chainId = mappedLoanAssetChainId;
    _metadata.loanAsset = mappedLoanAsset;

    mappedLoanAssets[satelliteLoanAssetChainId][satelliteLoanAsset] = _metadata;

    emit SatelliteLoanMarketSupported(
        satelliteLoanAsset,
        satelliteLoanAssetChainId,
        mappedLoanAsset,
        mappedLoanAssetChainId
    );
}
```

Snippet 4.10: Definition of `MasterAdmin.supportSatelliteLoanMarket()`

`supportSatelliteLoanMarket` after some collateral is deposited to the collateral market, the `totalSupplied` of the loanMarket will be less than the actual amount of collateral that has been deposited into the protocol. This may cause the liquidity check to fail, resulting in a revert and preventing the user from withdrawing their funds. As a concrete example, consider the following scenario:

1. Initial State: `supportMarket` is called to enable a PToken with underlying token A. This PToken does not have an associated loan asset.
2. Alice deposits 100 of token A.
3. Protocol then adds a new loan asset "B" and calls `supportSatelliteLoanMarket()` to associate the PToken with B.
4. Alice tries to withdraw 1 of their token A. Because the loan asset was newly added, the values of both `totalSupplied` and `totalBorrows` are zero.
5. The condition `totalBorrows > totalSupplied` will be false, since the `totalSupplied` is zero. However, `(totalSupplied - totalBorrows) < actualWithdrawAmount` will evaluate to true.
6. Thus, the liquidity check will fail, despite (1) the user having a sufficient PToken/collateral balance, and (2) the protocol having enough collateral to cover the withdrawal.

A similar liquidity check in `_borrowAllowed()` (which is called when a user initiates a borrow request) may also fail.

Recommendation The developers should insert additional logic to ensure that the `totalSupplied` of a loan market is consistent with the actual amount deposited.

```
LoanMarketMetadata memory loanMarket = mappedLoanAssets[pTokenChainId][pToken];
uint256 totalSupplied = loanMarkets[loanMarket.loanAsset][loanMarket.chainId].
    totalSupplied;
if (totalSupplied != type(uint256).max) {
    uint256 totalBorrows = loanMarkets[loanMarket.loanAsset][loanMarket.chainId].
    totalBorrows;
    if (totalBorrows > totalSupplied || (totalSupplied - totalBorrows) <
    pTokenWithdrawAmount) {
        revert InsufficientLiquidity();
    }
    loanMarkets[loanMarket.loanAsset][loanMarket.chainId].totalSupplied -=
    actualWithdrawAmount;
}
```

Snippet 4.11: Relevant lines in `_withdrawAllowed()`

```
if (loanMarkets[targetMarket.loanAsset][targetMarket.chainId].totalBorrows +
    borrowAmount >
    loanMarkets[targetMarket.loanAsset][targetMarket.chainId].totalSupplied
) {
    revert InsufficientLiquidity();
}
```

Snippet 4.12: Similar liquidity check in `_borrowAllowed()`. If `totalSupplied` undercounts the actual amount of collateral available, it is likely for this comparison to evaluate to true, causing a revert.

Developer Response The developers updated the `supportLoanMarket()` method so that it initializes the `totalSupplied` value when a loan market is first created.

4.1.7 V-PRI2-VUL-007: Rounding error may cause Aave PToken withdraw to revert

Severity	Medium	**Commit**	b1ee399
Type	Denial of Service	**Status**	Fixed
File(s)	`[...]/ASVTokenV3RewardsController.sol`		
Location(s)	_withdrawRewards()		

While withdrawing rewards from the Aave PToken, the protocol needs to calculate reward per PToken using fixed point arithmetic. This exchange rate is then passed to `_queryUseRewardsBalance()` to calculate how much reward a user should get. The exchange rate is calculated as follows:

- `marketRewardsBalance` stores the total reward balance queried from the collateral token, using the precision of the underlying reward tokens tracked by Aave.
- `rewardFactor` stores the precision scale factor for the `marketRewardsBalance`.
- `marketPTokenTotalSupply` stores the total number of PTokens that have been deposited into the contract.
- `pTokenFactor` stores the precision factor for the `marketPTokenTotalSupply`. It is the number of decimals in the PToken contract.

Then the `marketRewardsExchangeRate` is calculated using

`marketRewardsExchangeRate = marketRewardsBalance * pTokenFactor / (marketPTokenTotalSupply * rewardFactor)` However, note that all units in this multiplication will be eliminated, meaning

```
uint256 marketRewardsBalance = _queryMarketRewardsBalance(rewardAddress);
uint256 marketPTokenTotalSupply = AavePTokenStorage(address(this)).totalSupply();

uint256 rewardFactor = 10**ERC20(rewardAddress).decimals();
uint256 pTokenFactor = 10**PTokenStorage(address(this)).decimals();
uint256 marketRewardsExchangeRate;

if (marketPTokenTotalSupply != 0) {
    marketRewardsExchangeRate = marketRewardsBalance * pTokenFactor / (
    marketPTokenTotalSupply * rewardFactor);
}

uint256 userRewardsBalance = _queryUserRewardsBalance(
    user,
    rewardAddress,
    marketRewardsExchangeRate
);
```

Snippet 4.13: Lines in `_withdrawRewards` that calculate the exchange rate

that the resulting scale factor is only one. This may lead to values being rounded down. For example, if the reward token precision is 18 decimals, the PToken precision is 9 decimals, `marketRewardsBalance = 9 * 10**17` (0.9 reward tokens), and `marketPTokenTotalSupply = 2 ** 10^9` (2 collateral tokens), then `marketRewardsExchangeRate` will be 0.

When `marketRewardsExchangeRate` is small, `_queryUserRewardsBalanace` is likely to revert due to a check that ensures that the reward exchange rate that the user last withdrew rewards at is strictly larger than `marketRewardsExchangeRate`.

```
function _queryUserRewardsBalance(
    address user,
    address rewardAddress,
    uint256 marketRewardsExchangeRate
) internal override view returns (uint256 userRewardsBalance) {
    if (user == address(0) || rewardAddress == address(0)) revert AddressExpected();

    uint256 userRewardsExchangeRate = userRewards[user][rewardAddress].exchangeRate;
    uint256 userPTokenBalance = AavePTokenStorage(address(this)).accountTokens(user);

    if (userRewardsExchangeRate > marketRewardsExchangeRate) revert InvalidExchangeRate();
```

Snippet 4.14: Lines in `_queryUserRewardsBalance()` leading to revert

Impact Because `_withdrawRewards()` is called by the deposit, withdraw, and liquidation logic, this may cause any of those functions to revert. Furthermore, any user that deposits funds is likely to increase the probability of the rounding error occurring as they will be decreasing the numerator and increasing the denominator of the division.

Recommendation The developers should also multiply by a constant such as `10**FACTOR_DECIMALS` when computing `marketRewardsExchangeRate`. However, note that this `10**FACTOR_DECIMALS` is only an *example*; the developer should make sure they multiply by a constant that results in the correct precision and units.

Developer Response The developers will be removing this contract from the code base as they do not plan to deploy it.

4.1.8 V-PRI2-VUL-008: RequestController.deposit does not forward msg.value

Severity	Medium	Commit	b1ee399
Type	Logic Error	Status	Fixed
File(s)	satellite/requestController/RequestController.sol		
Location(s)	deposit()		

The `RequestController.deposit()` method allows users to deposit collateral into a given `PToken` address on the same satellite chain. The method is implemented by forwarding the call to `PToken.depositBehalf()`. The latter method requires its caller to send native currency (e.g., ether), which will be used as a gas fee for the cross-chain bridge that will deliver the deposit request.

Due to the way that `PToken.depositBehalf()` is invoked, the native currency will be sent to the `RequestController` contract, but the native currency will *not* be forwarded to the `PToken` contract for use in `depositBehalf()`.

```
function deposit(
    address route,
    address user,
    uint256 amount,
    address pTokenAddress
) external override payable virtual {
    if (pTokenAddress == address(0)) revert AddressExpected();
    if (user == address(0)) revert AddressExpected();

    IPToken(pTokenAddress).depositBehalf(route, user, amount);
}
```

Snippet 4.15: Implementation of `RequestController.deposit()`

Impact Any call to `RequestController.deposit()` will likely revert, as calling `PToken.depositBehalf()` in this way will likely revert.

- If the PToken's underlying collateral is address 0 (corresponding to native currency), then `_doTransferIn` will revert when it compares `msg.value` with the `amount` argument.
- Otherwise, the collateral will be transferred from the user to the PToken contract, and zero gas will be provided to the cross-chain bridge.
  - This will revert if the bridge requires a gas fee to be paid in native currency.
  - If no gas fee is required, then the message may be sent, and the transaction will be successful. In this case, the native currency will be locked in the `RequestController` contract.

Users can work around this issue by invoking `IPToken.deposit()` or `IPToken.depositBehalf()` directly instead of through the `RequestController`.

```
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 4.16: The lines in `PToken.depositBehalf()` that use the forwarded native currency.

Recommendation The line that invokes `IPToken.depositBehalf()` should be changed to forward the `msg.value`:

```
    IPToken(pTokenAddress).depositBehalf{value: msg.value}(route, user, amount);
```


4.1.9 V-PRI2-VUL-009: Liquidating loan asset of zero locks native currency collateral

Severity	Medium	**Commit**	b1ee399
Type	Locked Funds	**Status**	Fixed
File(s)	satellite/requestController/RequestController.sol		
Location(s)	liquidate()		

The `liquidate()` function can be invoked by a user (which we will call the "liquidator") to liquidate loan assets of another user (the `borrower`) that is backed by an insufficient amount of collateral. In this flow, the liquidator must pay back (partially or in full) the loan amount in order to receive the borrower's collateral. The liquidator must also pay native currency to the `liquidate()` function, which will be used as a gas fee for cross-chain messages.

In the implementation of `liquidate()`, it is possible for the liquidator to specify a `loanAsset` of `address(0)`, which seems strange given that (1) a `loanAsset` is typically a smart contract; and (2) other methods in `RequestController` that deal with loan assets assume that the `loanAsset` address is nonzero.

```
function liquidate(
    address route,
    address seizeToken, // asset the liquidator will be repaid on
    uint256 seizeTokenChainId, // chainId of the tokens to seize
    address borrower, // address of the user being liquidated
    address loanAsset, // asset to be repaid on local chain
    uint256 repayAmount // amount of asset to be repaid by liquidator right now on
    local chain
) external payable /* override */ {
    if (repayAmount == 0) revert ExpectedRepayAmount();

    if (loanAsset != address(0)) {
        ILoanAsset(loanAsset).processRepay(msg.sender, repayAmount);
    } else {
        if (msg.value < repayAmount) revert ExpectedValue();
        payable(loanAsset).transfer(repayAmount);
    }

    // send the liquidation
    _sendLiquidation(
        borrower,
        route,
        seizeToken,
        seizeTokenChainId,
        loanAsset,
        repayAmount
    );
}
```

Snippet 4.17: Implementation of `liquidate()`

Impact Assuming that the `loanAsset` is the zero address, `repayAmount` of native currency will be sent to the zero address, and neither the protocol nor the liquidator will not be able to retrieve that native currency.

Developer Response The developers noted that `loanAsset` is expected to be nonzero. They updated the code so that it always invokes `processRepay()`; this causes a revert if a zero address is provided.

4.1.10 V-PRI2-VUL-010: Potential rounding error causes revert in queryUserRewardsBalance

Severity	Low	Commit	b1ee399
Type	Denial of Service	Status	Fixed
File(s)	`[...]/ASVTokenV3RewardsController.sol`		
Location(s)	queryUserRewardsBalance()		

Similar to V-PRI2-VUL-007, a missing precision factor in a multiplication in `queryUserRewardsBalance()` may cause the function to revert.

```
function queryUserRewardsBalance(
    address user,
    address rewardAddress
) external override view returns (uint256) {
    uint256 marketRewardsBalance = _queryMarketRewardsBalance(rewardAddress);
    uint256 marketPTokenTotalSupply = AavePTokenStorage(address(this)).totalSupply();

    uint256 marketRewardsExchangeRate;

    if (marketPTokenTotalSupply != 0) {
        uint256 rewardFactor = 10**ERC20(rewardAddress).decimals();
        uint256 pTokenFactor = 10**PTokenStorage(address(this)).decimals();

        marketRewardsExchangeRate = marketRewardsBalance * pTokenFactor / (
    marketPTokenTotalSupply * rewardFactor);
    }

    return _queryUserRewardsBalance(user, rewardAddress, marketRewardsExchangeRate);
}
```

Snippet 4.18: Implementation of `queryUserRewardsBalance()`

Impact Calls to `queryUserRewardsBalance()` may revert unexpectedly.

Recommendation The developers should also multiply by a constant such as `10**FACTOR_DECIMALS` when computing `marketRewardsExchangeRate`. However, note that this `10**FACTOR_DECIMALS` is only an *example*; the developer should make sure they multiply by a constant that results in the correct precision and units.

Developer Response The developers will be removing this contract from the code base as they do not plan to deploy it.

```
function _queryUserRewardsBalance(
    address user,
    address rewardAddress,
    uint256 marketRewardsExchangeRate
) internal override view returns (uint256 userRewardsBalance) {
    if (user == address(0) || rewardAddress == address(0)) revert AddressExpected();

    uint256 userRewardsExchangeRate = userRewards[user][rewardAddress].exchangeRate;
    uint256 userPTokenBalance = AavePTokenStorage(address(this)).accountTokens(user);

    if (userRewardsExchangeRate > marketRewardsExchangeRate) revert
    InvalidExchangeRate();
```

Snippet 4.19: Lines in `_queryUserRewardsBalance()` leading to revert

4.1.11 V-PRI2-VUL-011: changeProtocolIncentive does not validate bounds

Severity	Low	**Commit**	b1ee399
Type	Data Validation	**Status**	Fixed
File(s)	`[...]/ASVTokenV3RewardsController.sol`		
Location(s)	_withdrawRewards()		

The `protocolIncentive` variable in the rewards controller defines the percentage of rewards that should be deducted from the user's redeemed rewards and given to the admins of the protocol. In `ASVTokenV3RewardsController._withdrawRewards()`, it is assumed that `protocolIncentive` is at most `10**FACTOR_DECIMALS`; however, this property is not enforced in the `changeProtocolIncentive()` method, where an admin sets the `protocolIncentive` variable.

```
uint256 protocolShare = claimedRewards * protocolIncentive / FACTOR_DECIMALS;
uint256 adjustedClaimedRewards = claimedRewards - protocolShare;
```

Snippet 4.20: Lines where `protocolIncentive` is used in `ASVTokenV3RewardsController._withdrawRewards()`

```
function changeProtocolIncentive(
    uint256 newProtocolIncentive
) external override {
    if (msg.sender != PToken(address(this)).admin()) revert OnlyAdmin();

    emit ChangeProtocolIncentive(protocolIncentive, newProtocolIncentive);

    protocolIncentive = newProtocolIncentive;
}
```

Snippet 4.21: Definition of `RewardsControllerAdmin.changeProtocolIncentive()`

Impact If `protocolIncentive` is set to a value larger than `10**FACTOR_DECIMALS`, then `AavePToken.withdraw()` will always revert.

Recommendation Add a check in `changeProtocolIncentive()` that validates that `newProtocolIncentive <= 10**FACTOR_DECIMALS`.

Developer Response The developers will be removing this contract from the code base as they do not plan to deploy it.

4.1.12 V-PRI2-VUL-012: withdrawReserves refunds gas to wrong account

Severity	Low	Commit	b1ee399
Type	Logic Error	Status	Fixed
File(s)	master/MasterAdmin.sol		
Location(s)	withdrawReserves()		

The `withdrawReserves()` method allows an admin to withdraw the protocol's reserves from a loan market to a given `receiver` address on the loan asset's chain. This is done by constructing a `FBBorrow` packet and sending it to the target loan asset through the middle layer contract. However, when sending the packet, the gas refund address is set to the `receiver` and not the `msg.sender` that pays for the gas.

```
function withdrawReserves(
    uint256 withdrawAmount,
    address loanAsset,
    uint256 loanAssetChainId,
    address receiver
) external payable onlyAdmin() {
    if (receiver == address(0)) revert AddressExpected();
    // ...
    // We call FBBorrow flow so we can generalize the codepath for this flow. It will
     allow a "withdraw" of a given loanAsset
    bytes memory payload = abi.encode(
        IHelper.FBBorrow({
            metadata: uint256(0),
            selector: FB_BORROW,
            user: receiver,
            borrowAmount: withdrawAmount,
            loanAsset: loanAsset
        })
    );

    middleLayer.msend{ value: msg.value }(
        loanAssetChainId,
        payload, // bytes payload
        payable(receiver), // refund address
        true
    );
```

Snippet 4.22: Relevant lines in `withdrawReserves()`

Impact The gas refund will be sent to the receiver address instead of to the calling admin. Furthermore, the receiver may not be on the same chain as the master contract, in which case the gas refund may be sent to a completely incorrect address.

Recommendation Change the refund address to `msg.sender`.

4.1.13 V-PRI2-VUL-013: PToken does not validate decimals of underlying

Severity	Low	**Commit**	b1ee399
Type	Data Validation	**Status**	Fixed
File(s)	satellite/pToken/implementations		
Location(s)	initialize()		

A PToken contract wraps an ERC20 token ("underlying collateral") to allow them to be used on the protocol's money market. Each PToken contains a `decimals` storage variable that tracks the precision used for the token balances. Based on a discussion with the developers, the `decimals` variable is assumed to be equal to the value of `.decimals()` on the underlying collateral token; however, there is no validation logic that enforces this assumption.

```
function initialize(
    address _underlying,
    address _middleLayer,
    uint256 _masterCID,
    uint8 _decimals
) external payable initializer() {
    __UUPSUpgradeable_init();

    if (address(_middleLayer) == address(0)) revert AddressExpected();

    if (_decimals == 0 || _masterCID == 0) revert ParamOutOfBounds();

    underlying = _underlying;
    middleLayer = IMiddleLayer(_middleLayer);
    masterCID = _masterCID;
    decimals = _decimals;

    admin = payable(msg.sender);
}
```

Snippet 4.23: Example of where `decimals` is set in the base `PToken` implementation

Impact An admin could mistakenly set the decimals to a value other than the underlying token decimals, which would cause PToken amounts to be wrong in all calculations in all PToken methods. Since these amounts are sent to the master state, this could also propagate errors there.

Recommendation The `decimals` variable should be set with `decimals = IERC20(underlying).decimals()`, or by adding a `require` statement that checks that the `_decimals` parameter is equal to the underlying token's decimals. To ensure that all implementations have their logic updated, it would be good to also fix V-PRI2-VUL-020.

Developer Response The developers noted that the deployment scripts will ensure that a PToken is initialized with the same `decimals` value as the underlying token. However, the developers will add extra validation, since it is possible for an admin action (such as a DAO vote) to set decimals so that it does not match the underlying token's decimals.

Notes on Fix The developers changed the code to use the underlying decimals if the underlying is a nonzero address. If the underlying is the zero address (i.e., native currency), then decimals is assumed to be 18.

4.1.14 V-PRI2-VUL-014: Minor rounding error when calculating liquidation repay amount

Severity	Low	**Commit**	b1ee399
Type	Logic Error	**Status**	Fixed
File(s)	`master/MasterInternals.sol`		
Location(s)	_liquidateCalculateSeizeTokens()		

In the liquidation flow, a liquidator repays the loan borrowed by a delinquent borrower in exchange for a portion of or the full amount of collateral put up by the borrower. The protocol takes a percentage of the repayment amount as a fee before applying the repayment. This fee is calculated in `_liquidateCalculateSeizeTokens()`. There is a rounding issue that may

```
protocolSeizeAmount = rawRepayAmount * protocolSeizeShare / 10**FACTOR_DECIMALS;
actualRepayAmount = rawRepayAmount * (10**FACTOR_DECIMALS - protocolSeizeShare) /
    10**FACTOR_DECIMALS;
```

Snippet 4.24: Snippet in `_liquidateCalculateSeizeTokens()` where the fee is calculated.

it possible for `protocolSeizeAmount + actualRepayAmount` to be *smaller* than `rawRepayAmount`, which is unexpected.

First, note that the fee corresponds to `protocolSeizeAmount`, which is the repayment amount provided by the liquidator (`rawRepayAmount`) multiplied by the fee percentage (`protocolSeizeShare`). The division by `10**FACTOR_DECIMALS` ensures that `protocolSeizeAmount` has the correct precision. However, note that the division will round down any numerator smaller than `10** FACTOR_DECIMALS` to zero.

The `actualRepayAmount`, which is the amount to actually count towards the repayment, is calculated as `(100 - seize share percentage)%` of the `rawRepayAmount`. This also suffers from the same rounding issue.

Impact Because both amounts are rounded down, there may be cases where a user provides a repay amount that is theoretically high enough to repay the loan, but in practice will be insufficient due to the rounding errors.

Recommendation Calculate `actualRepayAmount` as `rawRepayAmount - protocolSeizeAmount`. Amounts that are rounded down will contribute to `actualRepayAmount` , allowing repayments to go through in the scenario described above. As an additional benefit, this will save some gas.

4.1.15 V-PRI2-VUL-015: supportMarket can be called on a previously listed market

Severity	Low	Commit	b1ee399
Type	Data Validation	Status	Fixed
File(s)	master/MasterAdmin.sol		
Location(s)	supportMarket(), listCollateralMarket()		

Given a PToken, an admin can call `supportMarket()` to list a new collateral market for that PToken. This will set various parameters such as the liquidity incentive, the number of decimals used for the PToken, etc. To avoid a market from being listed twice (e.g., by accident), the function will revert if `isListed` flag is already set. However, it is still possible to list a market

```
function supportMarket(
    address pToken,
    uint256 chainId,
    uint8 decimals,
    address underlying,
    uint256 liquidityIncentive,
    uint256 protocolSeizeShare,
    bool isRebase
) external override onlyAdmin() {
    if (markets[chainId][pToken].isListed) revert MarketExists();
    if (pToken == address(0)) revert AddressExpected();
    if (liquidityIncentive > 10**FACTOR_DECIMALS) revert InvalidPrecision();
    if (protocolSeizeShare > liquidityIncentive) revert InvalidProtocolSeizeShare();
    // ...
    markets[chainId][pToken].isListed = true;
    // ...
    collateralValueIndex.push(
        MarketIndex({
            pToken: pToken,
            chainId: chainId,
            marketType: CollateralMarketType.NoModify
        })
    );
```

Snippet 4.25: Relevant lines in `supportMarket()`

twice. If the admin uses the `listCollateralMarket()` to set the `isListed` flag to false and thereby *unlist* the market, then the admin will be able to invoke `supportMarket()` again.

Impact If `supportMarket()` is called on a PToken twice, then it will be inserted into `collateralValueIndex` twice. Since the entries of `collateralValueIndex` are used to compute collateral and loan amounts, this means that the PToken (or its underlying) amounts will be double-counted.

Recommendation In addition to using `isListed`, check that some other entry of `supportMarket()` must be nonzero. For example, since `supportMarket()` can only be called with a nonzero PToken, requiring that the existing `markets` entry has a nonzero PToken will ensure that the method cannot be called twice.


```
function listCollateralMarket(
    uint256 chainId,
    address pToken,
    bool shouldList
) external override onlyAdmin() {
    if (pToken == address(0)) revert AddressExpected();

    markets[chainId][pToken].isListed = shouldList;

    emit CollateralMarketListed(chainId, pToken, shouldList);
}
```

Snippet 4.26: Definition of `listCollateralMarket()`

Developer Response The developers have changed the code to revert if the PToken is already contained in `collateralValueIndex` .

4.1.16 V-PRI2-VUL-016: Associated loan market not validated before use

Severity	Low	**Commit**	b1ee399
Type	Data Validation	**Status**	Fixed
File(s)	master/MasterInternals.sol		
Location(s)	_deposit(), _withdrawAllowed()		

A PToken can be "associated" with a loan asset when an admin calls `supportSatelliteLoanMarket()` on the PToken and loan asset. In markets created this way, the PToken's underlying token will be used as collateral for the loan asset. Such loan markets will be stored in the `mappedLoanAssets` mapping, which maps PTokens to their associated loan asset.

This `mappedLoanAssets` is used in `_deposit()` and `_withdrawAllowed()`. and is used to keep collateral deposit amounts in sync with the loan market collateral amounts. For example, whenever a user deposits collateral into a PToken that does not have a first-party loan asset, the master contract increases the amount of collateral deposited for the associated money market. However, if `supportSatelliteLoanMarket()` is *not* called on a PToken, then the `mappedLoanAssets`

```
{
    LoanMarketMetadata memory targetMarket = mappedLoanAssets[pTokenChainId][pToken];
    if (loanMarkets[targetMarket.loanAsset][targetMarket.chainId].totalSupplied !=
    type(uint256).max) {
        loanMarkets[targetMarket.loanAsset][targetMarket.chainId].totalSupplied +=
    depositAmount;
    }
}

markets[pTokenChainId][pToken].totalSupply += actualDepositAmount;
```

Snippet 4.27: Relevant code in `_deposit()`

entry for that PToken will be the zero value. In this case, the `loanMarket.loanAsset` and `loanMarket.chainId` will both be zero. Thus, all deposits to PTokens that do not have an associated `mappedLoanAssets` entry will increase the same entry `loanMarkets[0][0]`.

Impact At the time of the audit, we do not believe this issue has any impact besides wasted gas, as the extra storage operations to `loanMarkets[0][0]` may cost a nontrivial amount of gas.

We note that not validating the `mappedLoanAssets` entry may lead to issues in the future. In particular, the `_withdrawAllowed()` method uses the `totalSupplied` value in the following ways:

- Presently, the `totalSupplied` is used in the liquidity check, and the `loanMarkets[0][0]` entry could be increased by deposits from multiple PTokens, this means that `totalSupplied` is likely to be a very large value. Thus, `totalBorrows > totalSupplied` is likely to be false, and `(totalSupplied - totalBorrows) < pTokenWithdrawAmount` is also likely to be false. This means that the liquidity check will always succeed.
- The `totalSupplied` will always be larger than the `actualWithdrawAmount` due to how it is updated in both deposits and withdrawals, so a subtraction overflow cannot happen (and subsequently cause an unintended revert on a withdraw that should actually succeed).

While the overall logic in `_withdrawAllowed()` is not affected, a future change by the developers could cause any of the assumptions above to be invalidated. Such a change may make it possible, for example, for withdraws to be reverted inadvertently, or for withdraws to be approved when they should not.

```
LoanMarketMetadata memory loanMarket = mappedLoanAssets[pTokenChainId][pToken];
uint256 totalSupplied = loanMarkets[loanMarket.loanAsset][loanMarket.chainId].
    totalSupplied;
if (totalSupplied != type(uint256).max) {
    uint256 totalBorrows = loanMarkets[loanMarket.loanAsset][loanMarket.chainId].
    totalBorrows;

    if (totalBorrows > totalSupplied || (totalSupplied - totalBorrows) <
    pTokenWithdrawAmount) {
        revert InsufficientLiquidity();
    }

    loanMarkets[loanMarket.loanAsset][loanMarket.chainId].totalSupplied -=
    actualWithdrawAmount;
}

markets[pTokenChainId][pToken].totalSupply -= pTokenWithdrawAmount;
```

Snippet 4.28: Relevant code in `_withdrawAllowed()`

Recommendation The `totalSupplied != type(uint256).max` condition is too lax. To avoid future issues, the developers should also check that the loan market in the `mappedLoanAssets` entry is actually listed before using any of the loan market fields.

4.1.17 V-PRI2-VUL-017: Same hash hardcoded in two locations

Severity	Warning	**Commit**	b1ee399
Type	Maintainability	**Status**	Fixed
File(s)	`middleLayer/IMiddleLayer.sol`, `util/CommonModifiers.sol`		
Location(s)	IMiddleLayer.CONTRACT_ID		

The `isMiddleLayer` modifier, which is used in multiple contracts, is used to check whether a given address corresponds to a MiddleLayer contract. This is done by checking a hardcoded hash value `IMiddleLayer.CONTRACT_ID()` . However, this hardcoded hash value is duplicated in `isMiddleLayer` and `IMiddleLayer.CONTRACT_ID`.

```
modifier isMiddleLayer(address newMiddleLayer) {
    if (IMiddleLayer(newMiddleLayer).CONTRACT_ID() != keccak256("contracts/
    middleLayer/MiddleLayer.sol")) {
        revert MiddleLayerExpected();
    }
    _;
}
```

Snippet 4.29: Definition of modifier `isMiddleLayer` in `CommonModifiers`

```
abstract contract IMiddleLayer {

    bytes32 public constant CONTRACT_ID = keccak256("contracts/middleLayer/
    MiddleLayer.sol");
```

Snippet 4.30: Location in `IMiddleLayer` that defines `CONTRACT_ID`

Impact If the developers change the string passed to `keccack256`, they will need to update it in both places, or else `isMiddleLayer` may revert unexpectedly.

Recommendation The developers should move `IMiddleLayer.CONTRACT_ID` to a top-level constant, which they should then reuse in both `IMiddleLayer` and `CommonModifiers`. For example:

```
bytes32 constant MIDDLE_LAYER_CONTRACT_ID = keccak256("contracts/middleLayer/
    MiddleLayer.sol");

abstract contract IMiddleLayer {
    bytes32 public constant CONTRACT_ID = MIDDLE_LAYER_CONTRACT_ID;
    // ...
}
```

4.1.18 V-PRI2-VUL-018: Duplicated logic in PToken deposit, depositBehalf

Severity	Warning	Commit	b1ee399
Type	Maintainability	Status	Fixed
File(s)	satellite/pToken/implementations		
Location(s)	deposit(), depositBehalf()		

All PToken contracts inherit from `PTokenBase`, which provides virtual functions `deposit` and `depositBehalf` that may be overridden by specific implementations of PToken. These functions first validate the provided arguments, and then perform implementation specific actions. Second,

```
if (amount == 0) revert ExpectedDepositAmount();
if (isFrozen) revert MarketIsFrozen(address(this));
if (isdeprecated) revert MarketIsdeprecated(address(this));
```

Snippet 4.31: Example of the shared validation logic at the beginning of `deposit()` and `depositBehalf()`

we observed that most of the `deposit()` implementations are duplicates of `depositBehalf()` with the `user` argument replaced by `msg.sender`.

We noted duplication in the following contracts:

- `PTokenBase`
- `CompoundPToken`
- `RebasePToken`
- `AavePToken`

Impact If the developers intend to modify the PToken validation logic or add new PTokens, they will need to remember to ensure that all PToken implementations use the same validation logic. This is error-prone and could result in access control bugs.

Recommendation

- The developers can add a modifier or internal function in `PTokenBase` that implements common validation logic that can be shared between all PToken implementations.
- To reduce code duplication between `deposit()` and `depositBehalf()`, the developers can also implement `deposit()` in terms of `depositBehalf()`.

4.1.19 V-PRI2-VUL-019: Missing events in RewardControllerAdmin

Severity	Warning	Commit	b1ee399
Type	Missing/Incorrect Eve	Status	Fixed
File(s)	`[...]/RewardsControllerAdmin.sol`		
Location(s)	_addToRewardsList(), _removeFromRewardsList()		

The `RewardControllerAdmin` contract allows the admin to add or remove reward tokens. This functionality is implemented in the internal functions `_addToRewardsList` and `_removeFromRewardsList` . However, these two functions and their callers do not emit any events that log changes to the reward list.

Developer Response The developers will be removing this contract from the code base as they do not plan to deploy it.

4.1.20 V-PRI2-VUL-020: Duplicated initialization logic in PToken implementations

Severity	Warning	Commit	b1ee399
Type	Maintainability	Status	Fixed
File(s)	contracts/satellite/pToken/implementations		
Location(s)	initialize()		

All PToken contracts inherit from the `PTokenBase` contract. These contracts have an `initialize` function that validates and sets various contract variables like `middleLayer` and `underlying` .

In many of the PToken implementation contracts, the `initialize()` logic is mostly duplicated (with some slight modifications). Some of the affected contracts are:

- `PToken`
- `AavePToken`
- `CompoundPToken`
- `RebasePToken`

```
function initialize(
        address _underlying,
        address _middleLayer,
        uint256 _masterCID,
        uint8 _decimals,
        address _rewardsControllerThirdParty
    ) external payable initializer() {
    __UUPSUpgradeable_init();

    if (
        address(_middleLayer) == address(0) ||
        address(_underlying) == address(0)
    ) revert AddressExpected();

    if (_decimals == 0 || _masterCID == 0) revert ParamOutOfBounds();

    underlying = _underlying;
    middleLayer = IMiddleLayer(_middleLayer);
    masterCID = _masterCID;
    decimals = _decimals;
    rewardsControllerThirdParty = _rewardsControllerThirdParty;

    admin = payable(msg.sender);
}
```

Snippet 4.32: Implementation of `AavePToken.initialize()`. Except for a few lines of code, most of the code is similar to `PToken.initialize()`.

Impact Because a large amount of the `initialize()` logic deals with validating the arguments, it is possible for developers to introduce bugs when adding new PTokens or modifying the validation logic in the future.

Recommendation Ensure that common initialization logic, especially related to data validation and access controls, are moved into a new internal method such as `__PToken__init()`. This internal method can be called from the `initialize()` method in each PToken implementation to ensure that common arguments are validated correctly.

4.1.21 V-PRI2-VUL-021: Buffer overflow in MiddleLayer._mreceive

Severity	Warning	**Commit**	b1ee399
Type	Logic Error	**Status**	Acknowledged
File(s)	middleLayer/MiddleLayer.sol		
Location(s)	_mreceive()		

The `MiddleLayer._mreceive()` function is used to decode incoming local or cross-chain messages and invoke corresponding method calls on the same-chain master or satellite chain contracts. Given a byte array `_payload` containing the contents of the message, `_mreceive()` determines which contract and which method to invoke by branching on the `selector` field of the packet. In most of the cases, the `extraData` is constructed by dynamically allocating a memory byte array and then manually setting the fields of the array. Finally, the actual call data is constructed by appending the `extraData` to the packet data and modifying `_payload` in-place so that the selector is inserted in front.

However, the way that the `extraData` is appended to the packet data is by copying the `extraData` to the memory location immediately after `_payload`, which means that `extraData` can overwrite any memory that has been dynamically allocated after the allocation of `_payload` and before the call to `_mreceive()`.

```
bytes memory extraData;
address targetContract;

if (selector == MASTER_REPAY) {
    targetContract = address(masterState);
    assembly {
        extraData := mload(0x40)
        mstore(0x40, add(extraData, 0x40))
        mstore(extraData, 0x20)
        mstore(add(extraData, 0x20), _srcChainId)
    }
}
```

Snippet 4.33: Example of how extraData is constructed when handling a `MasterRepay` message.

```
if (extraData.length != 0) {
    assembly {
        let extraDataLen := mload(extraData)
        let offset := add(_payload, mload(_payload))
        for { let i := 0x20 } or(eq(i, extraDataLen), lt(i, extraDataLen)) { i := add
    (i, 0x20) } {
            mstore(add(offset, i), mload(add(extraData, i)))
        }
    }
}
```

Snippet 4.34: Snippet in `_mreceive()` that copies the extra call arguments to after the bounds of the `_payload` buffer.

Impact At the time of the audit, the buffer overflow has limited impact due to the following factors:

- The only allocation that occurs after the buffer overflow is that of a return buffer when the call fails. In this case, the contents of the return buffer are immediately overwritten.
- Any memory that is allocated before the call to `_mreceive()` is not used afterwards.
- No memory is read after each call to `_mreceive()`.

However, the developers should be aware that if a memory variable is allocated before the call to `_mreceive()` and is used afterwards, then that variable may contain arbitrary or unexpected contents.

Recommendation The developers should document this problem clearly to avoid potential problems in the future.

To reduce the attack surface, developers can avoid the buffer overflow by copying the call data arguments to a freshly allocated memory array, especially if they intend to continue extending the `MiddleLayer` contract in the future.

Developer Response The developers noted that they implemented the copy in this way to save gas, and that they do not plan on modifying the `MiddleLayer` contract to allocate additional memory. They will add documentation to warn future developers.

4.1.22 V-PRI2-VUL-022: Potentially incorrect cast in unlockLiquidationRefund

Severity	Warning	**Commit**	b1ee399
Type	Maintainability	**Status**	Fixed
File(s)	satellite/requestController/RequestControllerMessageHandler.sol		
Location(s)	unlockLiquidationRefund()		

When a liquidation request is sent back from the master chain to a satellite chain, the middle layer contract will forward the corresponding `SRefundLiquidator` packet to the request controller contract's `unlockLiquidationRefund()` method. This method will (1) cast the packet's `pToken` parameter to a loan asset, and then (2) invoke the loan asset's `receiveBorrow` method to In the

```
function unlockLiquidationRefund(
    IHelper.SRefundLiquidator memory params
) external payable override onlyMid() {
    ILoanAsset(params.pToken).receiveBorrow(params.liquidator, params.refundAmount);
```

Snippet 4.35: Location where `receiveBorrow` is invoked

request controller, the invokes the `receiveBorrow` method of the target loan asset contract will be invoked to mint loan asset tokens for the liquidator.

Because PTokens are not loan assets, this cast seems incorrect. However, the `pToken` parameter is actually a *loan asset*, as can be observed when tracing the message flow in reverse:

- The `SRefundLiquidator` packet is constructed in `MasterMessageHandler._satelliteRefundLiquidator()`, which is called by `MasterMessageHandler.masterLiquidationRequest()`. The `ptoken` field of the `SRefundLiquidator` comes from a `IHelper.MLiquidateBorrow` packet's `loanAsset` field.
- The `IHelper.MLiquidateBorrow` comes from `RequestControllerMessageHandler._sendLiquidation()`, which is called by the external function `RequestControllerMessageHandler.liquidate()`.
- The `RequestControllerMessageHandler.liquidate()` is invoked when a user initiates a liquidation request on a satellite chain. The caller provides a target `loanAsset` as an argument.

To add to the confusion, both `PToken` and `LoanAsset` define `receiveBorrow` as part of their interfaces.

Impact It is easy for developers to mistakenly believe that `params.pToken` is a PToken when it is actually a loan asset (which can be either a concrete PToken or LoanAsset contract), which can increase the chance of introducing bugs in the future. This is exacerbated by the fact that both `PTokenBase` and `LoanAsset` each define a `receiveBorrow` method with the same function signature. See related issue: V-PRI2-VUL-027

Recommendation The `.pToken` field of `SLiquidateBorrow` should be renamed to `.loanAsset`.

4.1.23 V-PRI2-VUL-023: Unfairness while withdrawing collateral in low-liquidity situations

Severity	Warning	**Commit**	b1ee399
Type	Usability Issue	**Status**	Intended Behavior
File(s)	`master/MasterInternals.sol`		
Location(s)	_withdrawAllowed()		

The `MasterInternals._withdrawAllowed()` method checks the approval of a collateral withdrawal initiated on a satellite chain. Part of the approval check is to determine whether the protocol has sufficient liquidity to cover the withdrawal; if the protocol lacks liquidity, then the `_withdrawAllowed()` method will revert.

Due to the way the `_withdrawAllowed()` function is implemented, the users who withdraw funds first will be able to withdraw their collateral, while users who withdraw later will have their withdraw requests reverted.

```
if (totalSupplied != type(uint256).max) {
    uint256 totalBorrows = loanMarkets[loanMarket.loanAsset][loanMarket.chainId].totalBorrows;
    if (totalBorrows > totalSupplied || (totalSupplied - totalBorrows) < pTokenWithdrawAmount) {
        revert InsufficientLiquidity();
    }

    loanMarkets[loanMarket.loanAsset][loanMarket.chainId].totalSupplied -= actualWithdrawAmount;
}
```

Snippet 4.36: Snippet in `_withdrawAllowed()` that performs the liquidity check.

Impact Users who are unable to withdraw their collateral will be very upset, and such a situation would be very damaging for the protocol.

Recommendation To mitigate scenarios in which there is insufficient liquidity, the developers could implement fairer withdraw mechanisms such as: withdrawal queues, pro-rata distributions of collateral, etc.

Developer Response The developers noted that this is a fairly standard practice, as found in other DeFi protocols. To avoid liquidity issues, the interest rate will be increased in low-liquidity situations in order to incentivize users to deposit collateral.

4.1.24 V-PRI2-VUL-024: Inconsistent comments in DoubleLinearIRMStorage

Severity	Warning	**Commit**	b1ee399
Type	Maintainability	**Status**	Fixed
File(s)	`[...]/DoubleLinearIRMStorage.sol`		
Location(s)	N/A		

The `DoubleLinearIRMStorage` contract defines storage variables used in the implementation of the double linear interest rate model. There are multiple comments on numerical variables that state that the variables are expressed in terms of "ray" (e.g., 10 to the power of 27). However, the `FACTOR_DECIMALS` constant is defined as 18, corresponding to "wad".

Impact If `FACTORS_DECIMAL` is supposed to be ray, then it should be set to 27; otherwise, multiple calculations performed in `DoubleLinearIRM` will be performed with incorrect precision.

Recommendation The developers should clarify whether the units are in wad or ray, and then they should make sure the comments are consistent with the implementation.

Developer Response The units should be in terms of wad.

4.1.25 V-PRI2-VUL-025: collateralBalances is confusingly named

Severity	Warning	**Commit**	b1ee399
Type	Maintainability	**Status**	Fixed
File(s)	`master/MasterStorage.sol`, `master/MasterInternals.sol`		
Location(s)	mapping collateralBalances, MasterInternals._collateralBalancesStored()		

The `collateralBalances` mapping keeps track of the amount of a given PToken that a given user has on a given chain. This is confusing, since the name suggests that it tracks the amount of underlying collateral, when it does not. Adding to the confusion, there is also a `_collateralBalancesStored()` method in `MasterInternals` which returns underlying collateral amounts instead of PToken amounts.

Impact The confusing names could increase the chance of the developers making mistakes and introducing bugs in the future.

Recommendation The developers should rename variables like `collateralBalances` and methods like `_collateralBalancesStored` to ensure that the terms are precise and unambiguous.

Developer Response The developers renamed the `collateralBalances` variable to `pTokenCollateralBalances`
.

4.1.26 V-PRI2-VUL-026: Liquidation response can forward msg.value twice

Severity	Warning	**Commit**	b1ee399
Type	Logic Error	**Status**	Acknowledged
File(s)	master/MasterMessageHandler.sol		
Location(s)	masterLiquidationRequest()		

When the master contract approves a liquidation request, it may send up to two messages in response. First, if the liquidator has overpaid and is due a refund, then a `SRefundLiquidator` message will be sent to the liquidator's chain to issue a refund to the liquidator. Second, a `SLiquidateBorrow` message will be sent to the PToken's chain to transfer the borrower's collateral to the liquidator. This is performed by two helper functions, `_satelliteRefundLiquidator` and `_satelliteLiquidateBorrow`, respectively. The implementation of each of these helper functions

```
if (refundAmount != 0) {
    _satelliteRefundLiquidator(
        chainId, //this is the chain id where liquidator funds are locked
        params.liquidator,
        refundAmount,
        params.loanAsset,
        seizeAmount
    );
    // ...
}

if (seizeAmount != 0) {
    _satelliteLiquidateBorrow(
        params.seizeToken,
        params.seizeTokenChainId,
        params.borrower,
        params.liquidator,
        seizeAmount
    );
    // ...
```

Snippet 4.37: Lines in `masterLiquidationRequest()` that sends the response.

involves constructing the message and then sending it through the middle layer contract. This further involves forwarding a cross-chain gas fee to the gateway (indicated by `value: msg.value`) that is paid for by the master contract. Thus, if the following conditions are met:

```
middleLayer.msend{value: msg.value}(
    seizeTokenChainId,
    payload, // bytes payload
    payable(liquidator), // refund address,
    true
);
```

Snippet 4.38: Example of how the `SRefundLiquidator` message is sent. The code for the `SLiquidateBorrow` is similar.

- Both `refundAmount` and `seizeAmount` are nonzero—which is possible if the liquidator overpays and has the request accepted.

- The master contract initially has zero native currency.
- `msg.value` is nonzero.

Then `msg.value` will be consumed when calling `_satelliteRefundLiquidator()`, causing the subsequent call to `_satelliteBorrowLiquidator()` to revert.

Impact We asked the developers what the intended behavior is, and they noted that the `MiddleLayer` will invoke master methods with a `msg.value` set to 0. Their test suite also confirms this fact. Thus, at the time of the audit, this issue does not have any impact. However, the developers should note that this will cause no gas fees to be provided to the route, which could cause future problems when they add integrations with additional third-party protocols.

Recommendation To avoid future confusion, the developers should document `masterLiquidationRequest()` as well as `MiddleLayer.mreceive()` to explain their assumptions about the `msg.value` .

4.1.27 V-PRI2-VUL-027: Implicit interface is shared by PToken and LoanAsset

Severity	Warning	**Commit**	b1ee399
Type	Maintainability	**Status**	Fixed
File(s)	`satellite/loanAsset/LoanAsset.sol`, `satellite/pToken/PTokenBase.sol`		
Location(s)	receiveBorrow(), processRepay()		

Users can borrow from a money market asset (implemented by a PToken contract) or a first-party loan asset (implemented by a LoanAsset contract). Both of these contracts implement two methods `receiveBorrow()` and `processRepay()` that have the same signature. However, in several

```
function receiveBorrow(
        address borrower,
        uint256 borrowAmount
    ) external /* override */ onlyRequestController() {
      if (borrowAmount == 0) revert AmountIsZero();

      _doTransferOut(borrower, underlying, borrowAmount);
  }
```

Snippet 4.39: `receiveBorrow()` in `PTokenBase.sol`

```
function receiveBorrow(
        address borrower,
        uint256 borrowAmount
  ) external onlyRequestController() {
      if (borrowAmount == 0) revert AmountIsZero();

      _mint(borrower, borrowAmount);
  }
```

Snippet 4.40: `receiveBorrow()` in `LoanAsset.sol`

places in contracts such as `RequestController`, an address may be cast to either a PToken or a LoanAsset to invoke the `receiveBorrow()` or `processRepay()` function, even if that address may actually not implement the interface that it is being cast to.

```
function borrowApproved(
    IHelper.FBBorrow memory params
) external payable override virtual onlyMid() {
    if (isLoanMarketFrozen[params.loanAsset]) revert MarketIsFrozen(params.loanAsset)
    ;

    ILoanAsset(params.loanAsset).receiveBorrow(params.user, params.borrowAmount);
```

Snippet 4.41: Example of how `receiveBorrow` might be called. This is from `RequestControllerMessageHandler.borrowApproved()`.

Impact Currently, the `RequestController` casts the given address to either an `IPToken` or `ILoanAsset` interface. This may lead the developer into think that the address may be a concrete

PToken or LoanAsset contract when it is not. Consequently, it is easier for a developer to call a function that is not actually defined in the casted contract.

Recommendation Define an interface that declares the `receiveBorrow()` and `processRepay()` functions. Both `LoanAsset` and `PToken` contract should extend from this interface, and the interface should be used in `RequestController` and `MiddleLayer` when the target of a `receiveBorrow()` or `processRepay()` can be either a `LoanAsset` or a `PToken`.

Developer Response The developers created an interface called `ILendable` and changed both `LoanAsset` and `PToken` to implement the interface.

4.1.28 V-PRI2-VUL-028: Missing events on interest accrual

Severity	Warning	**Commit**	b1ee399
Type	Missing/Incorrect Eve	**Status**	Fixed
File(s)	`master/MasterInternals.sol`		
Location(s)	_accrueInterestOnSingleLoanMarket()		

The protocol updates the interest rate of a given loan market in the `_accrueInterestOnSingleLoanMarket()` function. This function does not emit any event when the interests are updated.

Impact Due to the missing events, it may be harder to monitor loan market interest rate changes at runtime. Note that a loan market interest rate update also occurs when a user borrows or repays a loan, or when a user is liquidated.

Recommendation Emit an event when a loan market's interest rate is updated.

4.1.29 V-PRI2-VUL-029: Consider documenting units in calculations

Severity	Warning	**Commit**	b1ee399
Type	Maintainability	**Status**	Acknowledged
File(s)	N/A		
Location(s)	N/A		

The protocol frequently needs to perform calculations over different units and fixed point numbers with varying scales. For example, the protocol handles quantities such as:

- Various underlying collateral tokens, each with their own fixed point precision scales
- Various PTokens, each with their fixed point precision scales
- US$ value of tokens

These units and precision scales are not clearly documented in the code.

Impact Due to the lack of documentation, it is easier for the developers to introduce unit conversion and integer precision errors such as V-PRI2-VUL-001and V-PRI2-VUL-002as they continue to modify the source code.

Recommendation The developers should clearly document the units and the scales of all storage variables, function arguments, and function return values. Furthermore, the developers should also insert comments that indicate the expected units and precision scales before and after calculations that require unit conversions. While this requires some up-front effort, it can help reduce the number of bugs introduced in the future.

