Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Prime Prime Protocol

Veridise Inc.
October 21, 2022

► **Prepared For:**

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

► **Prepared By:**

Jon Stephens
Bryan Tan
Kostas Ferles
Benjamin Mariano
Xiangan He

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

► **Version History:**

October 7, 2022	V1
October 21, 2022	V2

Contents

Executive Summary 1

From September 1 to September 21, Prime engaged Veridise to review the security of their Prime Protocol. The review covered the on-chain contracts that implement the protocol logic. Veridise conducted the assessment over 9 person-weeks, with 3 engineers reviewing code over 3 weeks from commit `706efa4` of the `CrossChainContracts` repository. The auditing strategy involved tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Summary of issues detected. The audit uncovered 17 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, bug V-PRI-VUL-001 can lead to dropped and corrupted cross-chain messages, while bug V-PRI-VUL-002 allows users to manipulate exchange rates by resending stale messages. The Veridise auditors also identified several moderate-severity issues, such as unchecked return values for critical functions (V-PRI-VUL-003), potentially locked funds (V-PRI-VUL-004, V-PRI-VUL-007) and possible price instability via unrestricted arbitrage (V-PRI-VUL-006). In addition to these concerns, auditors also identified a number of other concerns, including several missing input validations (V-PRI-VUL-010,V-PRI-VUL-011,V-PRI-VUL-012, V-PRI-VUL-013), unintentional truncation leading to incorrectly rounded values (V-PRI-VUL-009), as well as several code optimizations and maintainability suggestions (V-PRI-VUL-014, V-PRI-VUL-015, V-PRI-VUL-016, V-PRI-VUL-017).

Code assessment. The Prime Protocol implements a cross-chain lending-borrowing market. In particular, Prime Protocol enables users to deposit assets on any chain and receive an over-collateralized stablecoin loan backed by their entire portfolio of assets across all chains. Prime Protocol leverage a hub-and-spoke model, where a single master chain communicates with multiple satellite chains where users can borrow and repay loans. Like most lending protocols, Prime Protocol enables verified liquidators to pay off the debts of borrowers who have breached their maitenance margin in exchange for incentive. Prime Protocol relies on Axelar for secure communication between multiple blockchains and multiple Chainlink price feeds for stabilizing prices for assets cross-chain.

Prime provided the source code for the Prime Protocol contracts for review. A Typescript-based test-suite accompanied the source-code with tests written by the developers. These tests encompass fund deposit and withdrawal, oracle and bridge behavior, debt repayment, position liquidation, cross-chain message validity, home and satellite chain behaviors, collateral ratio and interest rate calculation. The test-suite also included a custom fuzzing mechanism built to further test elements of their protocol. Finally, the client provided extensive documentation as well as a whitepaper describing the goals of the exchange and intended behavior for the contracts.

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	`706efa4`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 29 - Oct. 7, 2022	Manual & Tools	3	9 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	2
Medium-Severity Issues	1	1
Low-Severity Issues	10	9
Warning-Severity Issues	4	2
Informational-Severity Issues	0	0
TOTAL	17	14

Table 2.4: Category Breakdown.

Name	Number
Logic Error	3
Locked Funds	4
Maintainability	4
Data Validation	5
Usability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the on-chain portion of the Prime Protocol. In our audit, we sought to answer the following questions:

- Are transactions between master and satellite chains always safe? Is state synchronicity maintained?
- Can messages be spoofed or replayed by attackers? What are the security guarantees of Axelar?
- Is it possible for a user to manipulate asset prices?
- Are admin-only functions restricted properly?
- Are users able to properly repay, borrow, and deposit funds?
- Are interest rates, incentives, etc. correctly calibrated to incentivize good behavior? Is decimal precision correctly maintained?

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 evaluate how the code behaves given unexpected inputs. To do this, we identified several code regions of interest and tested them against hand-written specifications.

Scope. This audit reviewed the on-chain behaviors of the Prime Protocol, including user behaviors on satellite chains, middle layer communication between satellite chains and the master, as well as master chain internal behaviors and liquidations. As such, Veridise auditors first reviewed the provided whitepaper and documentation to understand the desired behavior of the protocol as a whole. Then, the auditors inspected the provided tests to better understand the desired behavior of the provided contracts at a more granular level. Finally, auditors began a multi-week manual audit of the code assisted by both static analyzers and automated testing.

The audit focused on the following components in particular:

- Borrowing and lending logic.
- Cross-chain message sending and validation.
- Interest rates and incentives.

- Liquidation logic.
- Master state management and synchronicity of state across chains.

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

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-PRI-VUL-001	Incorrect rounding in ECC.roundPtr()	High	Fixed
V-PRI-VUL-002	Users can manipulate exchange rate	High	Fixed
V-PRI-VUL-003	masterDeposit() does not check result	Medium	Fixed
V-PRI-VUL-004	Locked funds due to no-op fallback and receive	Low	Fixed
V-PRI-VUL-005	onlySrc modifier bypassed if srcAddr is 0	Low	Fixed
V-PRI-VUL-006	No restriction on arbitrage amounts	Low	Acknowledged
V-PRI-VUL-007	Locked deposit funds	Low	Acknowledged
V-PRI-VUL-008	User liquidation difficulty	Low	Fixed
V-PRI-VUL-009	Unintentional truncation in tokensToDenom	Low	Fixed
V-PRI-VUL-010	No validation on liquidation	Low	Open
V-PRI-VUL-011	Checks recommendations for CRM	Low	Fixed
V-PRI-VUL-012	Checks recommendations for IRM	Low	Fixed
V-PRI-VUL-013	Checks recommendations for PrimeOracle	Low	Fixed
V-PRI-VUL-014	Use OpenZeppelin reentrancy guard	Warning	Open
V-PRI-VUL-015	MasterState.sol exceeding contract size limit	Warning	Acknowledged
V-PRI-VUL-016	Merge functions for clarity	Warning	Fixed
V-PRI-VUL-017	Axelar implementations should extend interfaces	Warning	Open

Note, for the statuses listed above, we define them as follows:

Fixed	Developers acknowledged the issue and added the fix suggested by Veridise auditors.
Addressed	Developers acknowledged the issue and added a different fix than the fix suggested by Veridise auditors. Veridise auditors verified that the fix performs as the customer expected.
Acknowledged	Developers acknowledged the issue but indicated that it either should be addressed (or already is) by code outside of the scope of the current audit or does not need to be addressed for other reasons at this time. If necessary, developers have forwarded the issue to the team/s in charge of the relevant code.
Open	Developers acknowledged the issue but have not yet addressed it.
Expected Behavior	Developers say the issue identified by Veridise auditors is actually the intended behavior of the code.

4.1 Detailed Description of Bugs

In this section, we describe each uncovered vulnerability in more detail.

4.1.1 V-PRI-VUL-001: Incorrect rounding in ECC.roundPtr()

Severity	High	**Commit**	706efa4
Type	Logic Error	**Status**	Fixed
Files	ecc/ECC.sol		
Functions	roundPtr		

Description The `roundPtr` method computes the value of `ptr` rounded to the closest multiple of `maxSize.slot` . However, the arithmetic for rounding up is incorrect: it should add `msze - delta` instead of `delta` .

```
function roundPtr(
    bytes32 ptr
) internal view returns (bytes32 /* ptr */) {
    assembly {
        let msze := sload(maxSize.slot)
        let delta := mod(ptr, msze)
        if gt(delta, 0) {
            let halfmsze := div(msze, 2)
            // round down at half
            if iszero(gt(delta, halfmsze)) { ptr := sub(ptr, delta) }
            if gt(delta, halfmsze) { ptr := add(ptr, delta) }
        }
    }
    return ptr;
}
```

Snippet 4.1: Implementation of `roundPtr`

Impact `roundPtr()` is intended to calculate a storage slot index that is aligned to multiples of `maxSize` slots (which is equal to 8 as of commit `fbba7e0`), and various functions in the `ECC` contract assume that `roundPtr()` indeed returns a slot index that is aligned. However, the bug above will cause the indices returned by `roundPtr()` to not be aligned.

Because the ECC module is in charge of recording, verifying, and resending messages, this error has significant potential consequences. In particular, this issue could result in (1) an inability to verify valid messages, (2) existing messages being overwritten or dropped, and (3) sending of bogus verified messages resulting from overwritten data on misalignment.

Recommendation

- In the `gt(delta, halfmsze)` case, change the assignment to `ptr := add(ptr, sub(msze, delta))`.
- Alternatively, the code can be changed so that the pointer is always rounded down. Probabilistically, the chance of a hash collision occurring when always rounding down is equal to that when rounding up 50% of the time and rounding down 50% of the time.

Developer Response The developers acknowledged the issue and took our recommendation to always round the pointer down in commit `eceb992`.

4.1.2 V-PRI-VUL-002: Users can manipulate Exchange Rate

Severity	High	**Commit**	706efa4
Type	Logic Error	**Status**	Fixed
Files	master/MasterMessageHandler.sol, ecc/ECC.sol		
Functions	masterDeposit, masterWithdraw, _getHypotheticalAccountLiquidity, resendMessage		

Description Currently in deposit and withdraw messages, satellites send along the current exchange rate for a loan asset. The master then uses the reported exchange rate to update the user's state on the master chain. In addition, the latest exchange rate is stored in the master state for later use when calculating a user's "account liquidity." While this gives satellites a greater ability to regulate themselves, it can also be abused by users with the help of `ECC.resendMessage` since the master does not check the age of a message.

In particular, users can take advantage of the lack of age checks by "pre-registering" a message. To do so a user submits a transaction that will send a message on the satellite but will revert on the master (note, it is also possible to do this by taking advantage of bridge downtimes). A user can therefore guarantee that the message can be resent without any further validation on the satellite and accepted again on the master (as a revert on master won't record the message in its ECC). Because old messages contain an outdated exchange rate, a malicious user can "pre-register" messages at a time when the exchange rate is desirable and resend them at a later time to make profit.

```
function withdraw(
    address route,
    uint256 withdrawAmount
) external override payable nonReentrant() {
    if (withdrawAmount == 0) revert ExpectedWithdrawAmount();
    if (totalSupply == 0) revert NothingToWithdraw();
    uint256 exchangeRate = _getExchangeRate();
    _sendWithdraw(
        msg.sender,
        route,
        withdrawAmount,
        exchangeRate
    );
}
```

Snippet 4.2: Checks performed on the satellite before sending a withdraw message

Impact There are two potential ways a user can use such an attack: (1) to pre-register transactions a user would eventually like to take or (2) to manipulate the functionality of the master state (such as `_getHypotheticalAccountLiquidity`).

In the first case, a user can pre-register withdraws with desirable (or manipulated) exchange rates. These withdraws can be registered before the user has even deposited funds in the protocol since the satellite does not validate the user's funds before sending a withdraw message. Later after the user has deposited funds and would like to withdraw them they can use `resendMessage` to perform a withdraw using the beneficial exchange rate.

In the second case, a user can pre-register a deposit or withdraw message specifically to manipulate the exchange rate. Similar to the previous case, a user would do so by first pre-registering messages with desirable (or manipulated) exchange rates. Later, a user can use this message to manipulate an account's hypothetical account liquidity. For example, a user could save a message with a good exchange rate to borrow more funds than the real exchange rate would allow. Alternatively, they could also save a message with a bad exchange rate to liquidate a user who's borrow position is still healthy.

Recommendation

- Use the timestamp included with resend messages in ECC to avoid resending stale messages. Checking first to see if a message is too old to send helps prevent users freely resending a pre-registered message.

Developer Response The developers acknowledged the issue and took our recommendation in commit `eceb992`.

4.1.3 V-PRI-VUL-003: masterDeposit() does not check result of _enterMarket()

Severity	Medium	**Commit**	706efa4
Type	Locked Funds	**Status**	Fixed
Files	master/MasterMessageHandler.sol		
Functions	masterDeposit		

Description Currently, calls to `masterDeposit` enter the user into a pToken market on a designated chain via a call to `_enterMarket`. As shown in the snippet below, the return value of `_enterMarket` is not checked when called in `masterDeposit`.

```
if (collateralBalances[chainId][params.user][params.pToken] == 0) {
    _enterMarket(params.pToken, chainId, params.user);

    emit NewCollateralBalance(params.user, chainId, params.pToken);
}
```

Snippet 4.3: Location in `masterDeposit()` where `_enterMarket()` is checked

The `_enterMarket` function has a return value indicating whether or not the market was entered successfully – failure to enter the market can occur if one attempts to enter an unlisted market. Part of the `_enterMarket` function is shown below.

```
if (accountMembership[borrower][chainId][pToken]) {
    return true;
}

if (!markets[chainId][pToken].isListed) {
    return false;
}
```

Snippet 4.4: Checks in `_enterMarket` that will be ignored due to this issue

Impact Failing to check this return value means that a deposit may incorrectly continue even if a user is attempting to enter an unlisted market. This could cause issues down the line when the user attempts to withdraw their deposit, as calls to `_exitMarket` may fail as the market was never correctly entered.

Recommendation Check the return result of `_enterMarket` and revert if the market is not successfully entered.

Developer Response The developers acknowledged the issue and took our recommendation in commit `eceb992`.

4.1.4 V-PRI-VUL-004: Locked funds due to no-op fallback and receive

Severity	Low	**Commit**	706efa4
Type	Locked Funds	**Status**	Fixed
Files	middleLayer/MiddleLayer.sol, loanAgent/LoanAgent.sol, loanAsset/LoanAsset.sol, pToken/PTokenBase.sol		
Functions	fallback, receive		

Description A number of contracts can receive ETH with no way for it to be retrieved from the contract. In particular, the following functions can lead to contracts receiving ETH which cannot be retrieved:

- The payable `MiddleLayer.msend()` method
- The `LoanAgent._sendRepay`, `LoanAgent._sendBorrow` internal methods invoked by external payable methods
- The `LoanAsset._sendTokensToChain` internal method invoked by external payable methods
- The `PTokenMessageHandler._sendDeposit`, `PTokenMessageHandler._sendWithdraw` internal methods invoked by external payable methods.

Impact Users could accidentally send funds to contracts which can never be retrieved, even by the contract owners.

Recommendation Remove the `fallback()` and `receive()` methods.

Developer Response The developers acknowledged the issue and took our recommendation in commit `eceb992`.

4.1.5 V-PRI-VUL-005: Can bypass onlySrc modifier if srcAddr is 0

Severity	Low	**Commit**	706efa4
Type	Data Validation	**Status**	Fixed
Files	middleLayer/routes/axelar/AxelarModifiers.sol		
Functions	onlySrc		

Description The middle layer validates the origin of a message by checking it against a set of known contract addresses on other blockchains. Due to how mappings are implemented, however, if `srcAddr` is ever `address(0)` , an attacker can bypass this check.

```
modifier onlySrc(uint256 srcChain, address srcAddr) {
    if (srcContracts[srcChain] != address(srcAddr)) revert OnlyAuth();
    _;
}
```

Snippet 4.5: The `onlySrc` implementation

Impact If a user is able to bypass this check, they would be able to process an arbitrary message on the message or satellite.

Recommendation While there should be no way for `srcAddr` to be zero if Axelar works as intended, the developers should be defensive in case a bug could be exploited in Axelar. Adding a zero address check would reduce the attack surface.

Developer Response The developers acknowledged the issue and took our recommendation in commit `eceb992`.

4.1.6 V-PRI-VUL-006: No Restrictions on arbitrage amounts

Severity	Low	**Commit**	706efa4
Type	Logic Error	**Status**	Acknowledged
Files	treasry/Treasury.sol		
Functions	mintLoanAsset, burnLoanAsset		

Description To maintain the stability of the Prime stablecoin (PUSD), the developers allow arbitrageurs to buy from and sell to the treasury directly to gain yields. Essentially, if the price of PUSD is too high, an arbitrageur can purchase PUSD from the treasury with alternative stablecoins. Similarly if the price of the prime stablecoin is too low, an arbitrageur can sell PUSD to the treasury in return for alternative stablecoins. While this can allow well-behaved users to stabilize PUSD's price, we are concerned that it could provide malicious users an avenue for profit since there is no restriction on the size of the purchase or sale. As such, a user may intentionally attempt to destabilize the price of PUSD so they may take advantage of the difference between an exchange's price and that of the treasury.

```
    function mintLoanAsset(
        address payable loanAsset,
        address tradeAsset,
        uint256 tradeAmount
    ) external payable override /* nonReentrant() */ returns (bool) {
        ...

        uint256 exchangeAmount = (tradeAmount * 10**loanAssetDecimals) / 10**
    tradeAssetDecimals;
        uint256 mintAmount = exchangeAmount * 10**FACTOR_DECIMALS / localLoanAsset[
    loanAsset][tradeAsset].mintPrice;

        assetReserves[tradeAsset] += tradeAmount;
        if (tradeAsset != address(0) && !_tradeAsset.transferFrom(msg.sender, address
    (this), tradeAmount)) revert TransferFailed(msg.sender, address(this));

        _loanAsset.mint(msg.sender, mintAmount);

        return true;
    }
```

Snippet 4.6: The treasury mint function that can mint an arbitrary number of tokens

As it stands, the treasury allows users to buy or sell arbitrary amounts of PUSD for a set price (`mintPrice/burnPrice` respectively) at the treasury. In addition, the treasury contract currently restricts these prices so they are between 1 and 1.05 so a user may always buy or sell at the treasury.

```
    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 4.7: Function that allows an asset to change the mint/burn price

Impact Since few restrictions exist on an arbitrageur, we are concerned that the treasury stability functions could be used to destabilize the price of PUSD for some gain. As of now, all attacks identified by Veridise engineers involve either (1) an attacker who is willing to lose a significant amount of money to destabilize PUSD or (2) a third party exchange containing major, exploitable flaws. Thus, we indicate this issue as low severity, nonetheless but strongly encourage the developers to protect against potential price instability.

Recommendation Restrict the number of tokens that can be minted or burnt either at once or over some period of time to be an adjustable percentage of the total supply.

Developer Response At this time, because Veridise engineers are unable to find a profitable attack and unlimited mints and burns may be desirable in some select scenarios, the Prime developers have opted not to address this issue.

4.1.7 V-PRI-VUL-007: Locked Deposit Funds

Severity	Low	**Commit**	706efa4
Type	Locked Funds	**Status**	Acknowledged
Files	PTokenBase.sol		
Functions	deposit		

Description Upon a deposit on the satellite, a PToken will perform some validation, collect the deposited funds, send a message to master and then mint the requested PTokens for the user as shown below.

```
function deposit(
    address route,
    uint256 amount
) external virtual override payable {
    if (isPaused) revert MarketIsPaused();
    if (amount == 0) revert ExpectedDepositAmount();
    uint256 exchangeRate = _getExchangeRate();
    uint256 actualTransferAmount = _doTransferIn(underlying, amount);
    uint256 actualDepositAmount = (actualTransferAmount * 10**EXCHANGE_RATE_DECIMALS)
     / exchangeRate;
    _sendDeposit(
        route,
        underlying == address(0)
            ? msg.value - actualDepositAmount
            : msg.value,
        actualDepositAmount,
        exchangeRate
    );
    totalSupply += actualDepositAmount;
    accountTokens[msg.sender] += actualDepositAmount;
}
```

Snippet 4.8: The deposit procedure for a PToken

Even though the PTokens have been minted, the given request may still be rejected on the master, namely if the target is paused on the master. In addition, as indicated by this issue we believe there may be further reasons for a rejection added in the future.

```
function masterDeposit(
    IHelper.MDeposit memory params,
    uint256 chainId
) external payable onlyMid() {
    // Do not accept new deposits on a paused market
    if (markets[chainId][params.pToken].isPaused) revert MarketIsPaused();
    ...
}
```

Snippet 4.9: Location where the master may reject a deposit message

Impact If this were to occur, a user may not be able to withdraw their funds from the PToken due to the fact that the deposit will not be recorded in the master state. Therefore, upon a withdrawal from the perspective of the master state these funds will not exist. To rectify this situation, the user must resend their message later once depositing is permitted (if it ever is). However, since a user will have received their PTokens when submitting the transaction it is unlikely they will know that this has occurred.

Recommendation In the current version of the protocol, the developers must ensure that the isPaused flags are consistent on the master and satellite. It is, however, unclear to us if both flags are necessary as only using a single paused flag will be less error prone. In addition, if the master may reject a deposit it might be necessary to award PTokens asynchronously similar to how a PToken performs a withdraw.

Developer Feedback Issues such as this are going to be addressed in the front-end. Essentially the front-end will monitor the events emitted from the satellite and master to find cases where an event fails on master and will prompt the user to take steps to address the issue (in cases where that's necessary).

4.1.8 V-PRI-VUL-008: User liquidation difficulty

Severity	Low	**Commit**	706efa4
Type	Usability	**Status**	Fixed
Files	master/MasterInternals.sol		
Functions	_liquidateBorrow		

Description When performing a liquidation, the protocol takes a percentage of the funds that are liquidated as a fee charged to the liquidator by subtracting it from the `repayAmount`. In addition, the protocol caps the given `repayAmount` at the size of the borrow being liquidated. A borrow can therefore only be fully liquidated by a liquidator if the protocol's fee is 0; otherwise, fully liquidating a borrow will likely require multiple liquidations.

```
function _liquidateBorrow(
    address pToken, address borrower, uint256 chainId,
    uint256 repayAmount, address route, address masterLoanMarket
) internal virtual override returns (bool) {
    ...

    /* We fetch the amount the borrower owes, with accumulated interest */
    uint256 accountBorrows = _borrowBalanceStored(
        borrower,
        masterLoanMarket
    );

    ...

    if (
        accountBorrows < repayAmount
    ) revert RepayTooMuch(repayAmount, accountBorrows);

    ...

    uint256 protocolSeizeShareAmount = (repayAmount * markets[chainId][pToken].
    protocolSeizeShare) / (10**FACTOR_DECIMALS);

    ...

    accountLoanMarketBorrows[borrower][masterLoanMarket].principal +=
    protocolSeizeShareAmount;
    loanMarkets[masterLoanMarket].totalBorrows += protocolSeizeShareAmount;

    ...
}
```

Snippet 4.10: Function _liquidateBorrow

Impact Once the borrow becomes small it may be ignored by liquidators as a liquidation will not yield as many profits. This could result in the protocol maintaining a number of small loans that are not properly collateralized.

Developer Response The developers acknowledged the issue and took our recommendation in commit `e732bf4`.

Recommendation Enable a liquidator to repay the full debt of the borrower plus the protocol fee so that a borrower can be completely liquidated in a single transaction.

4.1.9 V-PRI-VUL-009: Unintentional truncation in tokensToDenom calculation

Severity	Low	**Commit**	706efa4
Type	Data Validation	**Status**	Fixed
Files	MasterInternals.sol		
Functions	_getHypotheticalAccountLiquidity		

Description In the `tokensToDenom` calculation in `_getHypotheticalAccountLiquidity()`, an unintentional truncation currently causes the `tokensToDenom` value to be incorrectly calculated.

```
tokensToDenom = markets[_chainId][_pToken].exchangeRate * collateralFactor / 10**
    factorDecimals * oraclePrice / 10**oracleDecimals;
```

Snippet 4.11: 10**factorDecimals was placed before oraclePrice, resulting in a truncation

Impact This causes `collateralValue` , `sumCollateral`, and `sumBorrowPlusEffects` to be calculated potentially incorrectly, with `_getHypotheticalAccountLiquidity` being a component of the protocol's borrow, withdrawal and liquidation behavior (see MasterInternals.sol and MasterMessageHandler.sol).

Recommendation Adjust the order of operations to perform division after multiplication.

Developer Response The developers acknowledged the issue and took our recommendation in commit `7313d45`.

4.1.10 V-PRI-VUL-010: No validation on liquidation relationships

Severity	Low	**Commit**	706efa4
Type	Data Validation	**Status**	Open
Files	MasterInternals.sol		
Functions	_getHypotheticalAccountLiquidity		

Description The protocol currently has four correlated values that aid in a liquidation: `CRMRouterStorage.maintenanceLtvRatios`, `CRMRouterStorage.absMaxLtvRatios`, `MasterStorage.Market.protocolSeizeShare` and `MasterStorage.Market.liquidityIncentive`. First, to ensure that a user cannot be liquidated immediately after they borrow funds, the following relationship must hold:

```
1 > maintenanceLtvRatios[cid][collateral] >= absMaxLtvRatios[cid][collateral] > 0
```

In addition, to ensure that a user has enough collateral to ensure their loan can be fully liquidated with the liquidation bonus, the following relationship must hold:

```
1 / (1 + markets[cid][collateral].liquidationIncentive) > maintenanceLtvRatios[cid][
    collateral]
```

Finally, to ensure that a liquidation is sufficiently incentivized the following relationship must hold:

```
markets[cid][collateral].liquidationIncentive > markets[cid][collateral].
    protocolSeizeShare >= 0
```

Currently these relationships are not enforced in the protocol, which could result in an admin error.

Impact Particularly in the second case, an admin error could have drastic consequences for the protocol as technically the borrow would not be properly collateralized with respect to the incentives provided by the protocol. A liquidator could therefore purchase all of a user's collateral and leave the liquidated user with a borrow balance. Since the liquidated user has no collateral at this point liquidators don't have an incentive to liquidate the remainder of the borrow and the borrower doesn't have an incentive to repay their loan.

Recommendation Enforce the given relationships in code to remove the possibility of an admin error.

4.1.11 V-PRI-VUL-011: Checks recommendations for CRM

Severity	Low	**Commit**	706efa4
Type	Data Validation	**Status**	Fixed
Files	CRMAdmin.sol, CRM.sol		
Functions	setAbsMaxLtvRatio, getLoanMarketPremium		

Description The protocol currently has a few key functions in CRM related contracts that currently lack data validation in key spots. For `setAbsMaxLtvRatio`, a check should be added to ensure it is not accidentally set to 0.

```
    function setAbsMaxLtvRatio(
        uint256 chainId,
        address asset,
        uint256 maxLtvRatio
    ) external onlyAdmin() {
        absMaxLtvRatios[chainId][asset] = maxLtvRatio;
        emit AssetLtvRatioUpdated(chainId, asset, maxLtvRatio);
```

Snippet 4.12: setAbsMaxLtvRatio

In `getLoanMarketPremium`, ratio returned by oracle is assumed to be 1e18 here, but there is no validation that this is actually the case.

```
    function getLoanMarketPremium(
        address loanMarketOverlying,
        uint256 loanMarketUnderlyingChainId,
        address loanMarketUnderlying
    ) external view override returns (uint256, uint8) {
        (uint256 ratio, uint8 ratioDecimals) = primeOracle.getBorrowAssetExchangeRate
    (
            loanMarketOverlying,
            loanMarketUnderlyingChainId,
            loanMarketUnderlying
        );

        if (ratioDecimals < 4) revert InvalidPrecision();

        uint256 basePremium = 10**FACTOR_DECIMALS;
        int256 ratioDelta = int256(ratio) - int256(ratioFloor);

        if (ratio >= ratioCeiling) {
            return (basePremium, FACTOR_DECIMALS);
        } else if (ratioDelta <= int256(10**(ratioDecimals - 4))) { // INVARIANT:
    Premium should never be > 100x base premium
            return (100 * basePremium, FACTOR_DECIMALS);
        }
        return ((ratioCeiling - ratioFloor) * basePremium / uint256(ratioDelta),
    FACTOR_DECIMALS);
    }
```

Snippet 4.13: `getLoanMarketPremium` function

Impact If not checked, incorrect LTV ratios and loan market premiums can be applied to the protocol which could skew the incentive system towards bad behaviors.

Recommendation Implement checks as described above.

4.1.12 V-PRI-VUL-012: Checks recommendations for IRM

Severity	Low	**Commit**	706efa4
Type	Data Validation	**Status**	Fixed
Files	IRM.sol		
Functions	setBorrowRate		

Description Similar to V-PRI-VUL-011, ratio in `setBorrowRate` is assumed to be 1e18 here, but as a return value from the oracle, there should be validation that the ratio is returned as a 1e18 value. An enforcement on this assumption is recommended for data accuracy.

```
    function setBorrowRate(
        address loanMarketOverlying,
        uint256 loanMarketUnderlyingChainId,
        address loanMarketUnderlying
    ) external override onlyRouter() returns (uint256) {
        ...
        (uint256 ratio,) = primeOracle.getBorrowAssetExchangeRate(
            loanMarketOverlying,
            loanMarketUnderlyingChainId,
            loanMarketUnderlying
        );
        uint256 borrowInterestRateBaseline = borrowInterestRateBase;
        // increase interest above stable range
        if (ratio > upperTargetRatio)
        ...
    }
```

Snippet 4.14: The return value itself from getBorrowAssetExchangeRate is not checked here

Impact Similar to V-PRI-VUL-011, if these checks are violated, the borrow interest rate could be incorrectly assigned incentivizing undesired behavior from users.

Recommendation Implement checks as described above.

4.1.13 V-PRI-VUL-013: Checks recommendations for PrimeOracle

Severity	Low	**Commit**	706efa4
Type	Data Validation	**Status**	Fixed
Files	PrimeOracle.sol		
Functions	getUnderlyingPriceBorroww, getBorrowAssetExchangeRate		

In `getUnderlyingPriceBorrow`, the chainID and pUSDAddress are not verified. This assumes that the same pUSDAddress can be obtained on every satellite chain.

```
    function getUnderlyingPriceBorrow(
        uint256 chainId,
        address loanMarketUnderlying
    ) external view override returns (uint256, uint8) {

        if (loanMarketUnderlying == pusdAddress) {
            uint8 pusdDecimals = ERC20(pusdAddress).decimals();
            return (10**pusdDecimals, pusdDecimals);
        } else {
            return _getAssetPrice(chainId, loanMarketUnderlying);
        }
    }
```

Snippet 4.15: `getUnderlyingPriceBorrow` function

An additional check for revert should be implemented here to ensure that the secondary feed does not return 0 and cause logic errors in liquidation.

```
    function getBorrowAssetExchangeRate(
        address loanMarketOverlying,
        uint256 loanMarketUnderlyingChainId,
        address loanMarketUnderlying
    ) external view override returns (uint256 ratio, uint8 decimals) {
        if (loanMarketUnderlying == pusdAddress) {
            return _getAssetPrice(block.chainid, loanMarketUnderlying);
        }

        IPrimeOracleGetter primaryFeed =  primaryFeeds[loanMarketUnderlyingChainId][
    loanMarketUnderlying];
        if (address(primaryFeed) == address(0)) revert AddressExpected();
        (ratio, decimals) = primaryFeed.getAssetRatio(loanMarketOverlying,
    loanMarketUnderlying, loanMarketUnderlyingChainId);
        if (ratio == 0) {
            IPrimeOracleGetter secondaryFeed = primaryFeeds[
    loanMarketUnderlyingChainId][loanMarketUnderlying];
            if(address(secondaryFeed) == address(0)) revert AddressExpected();
            (ratio, decimals) = secondaryFeed.getAssetRatio(loanMarketOverlying,
    loanMarketUnderlying, loanMarketUnderlyingChainId);
        }
    }
```

Snippet 4.16: Secondary price feed could return 0 from function without revert.

Impact Checks on return values from oracles can, for example, prevent (1) the loan market premium to spike here and (2) cause the borrowInterestRatePerBlock to reach its maximum here.

Recommendation Track and check the `pusdAddress` for each satellite chain and make sure that in `getUnderlyingPriceBorrow` the `loanMarketUnderlying` is equal to this address. Also add revert in case where second price feed returns 0 in `getBorrowAssetExchangeRate`.

4.1.14 V-PRI-VUL-014: Optimization: Use battle-tested OpenZeppelin implementation

Severity	Warning	**Commit**	706efa4
Type	Maintainability	**Status**	Open
Files	CommonModifiers.sol		
Functions	modifier nonReentrant		

Description Currently, the protocol uses its own implementation of OpenZeppelin's Reentrancyguard. We recommend using the more gas-optimized and battle-tested OpenZeppelin implementation directly for better maintainability.

Impact Gas optimization

Recommendation Use OpenZeppelin's Reentrancy Guard.

4.1.15 V-PRI-VUL-015: MasterState.sol exceeding contract size limit

Severity	Warning	**Commit**	fbba7e0
Type	Maintainability	**Status**	Acknowledged
Files	master/MasterState.sol		
Functions	N/A		

Description A compilation warning that impacts mainnet deployment. Contracts may encounter deployment failures since it exceeds the size limit.

Recommendation Reduce contract size, or split the contract logic up into smaller contracts.

4.1.16 V-PRI-VUL-016: preProcessingValidation and flagMsgValidated should be combined

Severity	Warning	**Commit**	706efa4
Type	Maintainability	**Status**	Fixed
Files	ecc/ECC.sol		
Functions	preProcessingValidation, flagMsgValidated		

Description Currently two functions are used to pre-process a received message, `preProcessingValidation` and `flagMsgValidated` . However, since the safety of `flagMsgValidated` relies on `preProcessingValidation` occurring beforehand, the developer should consider combining these two functions.

Impact If in the future if `flagMsgValidated` was called without first calling `preProcessingValidation` the protocol could accept duplicate messages sent via `resendMessage` .

Recommendation Combine the two functions as mentioned.

Developer Response The developers acknowledged the issue and took our recommendation in commit `eceb992`.

4.1.17 V-PRI-VUL-017: Axelar implementations should extend Axelar interfaces

Severity	Warning	**Commit**	706efa4
Type	Maintainability	**Status**	Open
Files	middleLayer/routes/axelar/AxelarExecutable.sol		
Functions	N/A		

Description To ensure the implementation of AxelarExecutable matches the interface expected by Axelar, the developers should consider importing the version provided by Axelar rather than copying it.

Impact This ensures that the code is maintainable and that the accurate interface of Axelar is used to prevent any message-passing errors.

Recommendation Import the version of AxelarExecutable created by Axelar in production.