ackee blockchain

Prime

Protocol

by Ackee Blockchain

January 3, 2023

Contents

1. Document Revisions

1.0	Final report	November 25, 2022
1.1	Fix-review	December 7, 2022
1.2	Fix-review	January 3, 2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, Rockaway X.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Medium	-
	Low	Medium	Medium	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Stepan Sonsky	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Prime Protocol allows users to deposit assets on any supported chain and receive another asset loan backed by their entire portfolio of assets.

Revision 1.0

Prime engaged Ackee Blockchain to perform a security review of the Prime protocol with a total time donation of 52 engineering days in a period between September 26 and November 18, 2022 and the lead auditor was Jan Kalivoda.

The audit has been performed on the commit `7a602f0`.

The scope was full-repository excluding the following directories:

- contracts/util/dependency
- contracts/satellite/rewardsController
- contracts/master/staking

We began our review by using static analysis tools, namely Woke and Slither. Then we took a deep dive into the codebase and continued with hacking on a local deployment. Lastly, we were testing several scenarios with Brownie framework. During the review, we paid special attention to:

- ensuring the arithmetic of the system is correct,
- ensuring the correctness of the upgradeability mechanism,
- validating the correctness of data storing in the ECC contract and message resending,
- checking the multi-chain communication and possible chain id decoupling,
- checking the possibility of USP stablecoin misuse to hack the protocol,

- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

Our review resulted in 29 findings, ranging from Info to Medium severity.

In general, the project is solid. However, it is heavily dependent on the administrators (see Trust Model).

Ackee Blockchain recommends Prime:

- reconsider the usage of the anti-collision mechanism in the ECC contract,
- add more NatSpec comments to the code,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The fix review was done on November 30, 2022, on the given commit: `5adaf0b`.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 1.2

The fix review was done on January 3, 2023, on the given commit: `4264302`, and the client’s feedback for Revision 1.1. See Revision 1.2 for additional info.

The status of all reported issues has been updated and can be seen in the findings table.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Solution*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: USP can return different exchange rate	Medium	1.0	Fixed
M2: Duplicate routes can cause loss of funds	Medium	1.0	Fixed
M3: Admin role can be renounced	Medium	1.0	Fixed
M4: Two-phase Admin role transfer	Medium	1.0	Fixed
M5: The `setMidLayer` function has insufficient validation	Medium	1.0	Fixed
M6: CRM missing validations	Medium	1.0	Fixed

	Severity	Reported	Status
M7: IRM setters are not performing any kind of data validation	Medium	1.0	Fixed
M8: Unsafe transfers	Medium	1.0	Fixed
M9: Safe transfers are not checking for zero amounts	Medium	1.0	Fixed
M10: Duplicated balance values	Medium	1.0	Fixed
L1: Lack of project identifier for address validation	Low	1.0	Partially fixed
L2: The `liquidateCalculateSeizeTokens` is not checking for a valid PToken address	Low	1.0	Acknowledged
W1: Treasury allows to receive native tokens without minting	Warning	1.0	Fixed
W2: Hardcoded decimals for native tokens	Warning	1.0	Fixed
W3: Users can deposit but can not withdraw in a specific case	Warning	1.0	Fixed
W4: Inconsistent Master State values can break the calculations	Warning	1.0	Acknowledged
W5: Missing `initializer` modifier on the constructor	Warning	1.0	Fixed

	Severity	Reported	Status
W6: The `setBorrowRate` function does not emit events on different branching	Warning	1.0	Fixed
W7: Usage of `solc` optimizer	Warning	1.0	Acknowledged
W8: Lockfile overwriting	Warning	1.0	Fixed
I1: Inconsistent naming convention	Info	1.0	Acknowledged
I2: Misleading error for zero-address	Info	1.0	Acknowledged
I3: Commented out code	Info	1.0	Acknowledged
I4: Inconsistent usage of (pre/post)incrementation	Info	1.0	Fixed
I5: Unnecessary load	Info	1.0	Fixed
I6: LoanAgent code duplications	Info	1.0	Fixed
I7: ECC variables should be constants	Info	1.0	Fixed
I8: Abstract contracts naming	Info	1.0	Acknowledged
I9: Documentation	Info	1.0	Acknowledged

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Terms

Terms we find important for better understanding are described in the following sections.

Master chain

Master chain is the chain where MasterState is settled. It maintains all the information from its satellite chains and approves users' actions.

Satellite chain

Satellite chain is the chain where users interact with the protocol (except for liquidations). Actions from satellite chains are routed to the master chain for approvals.

Message-passing architecture

For message passing in the specified commit is used only Axelar. More information can be viewed in the Contracts and Actors sections.

Architecture

Prime Protocol allows users to deposit assets on any supported chain (Satellite chains) and receive a loan backed by their entire portfolio of assets. The decision if the user is allowed to borrow is performed on the Master chain. Communication between the Master chain and the Satellite chains is performed via message passing.

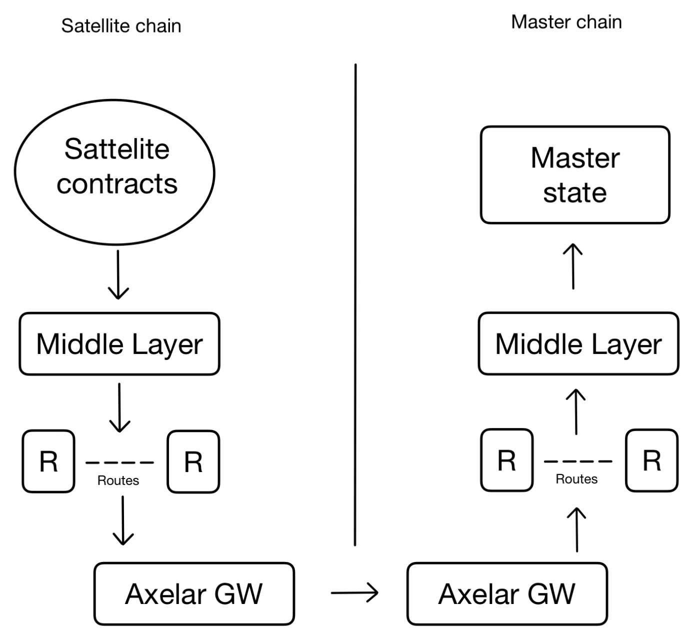

Figure 1. Simplified architecture of Prime Protocol

Contracts

Contracts we find important for better understanding are described in the following sections.

MasterState

`MasterState` is the central point of Prime Protocol and holds critical states from all satellite chains. Also, it performs calculations to validate key actions like borrowing, withdrawing collateral, or liquidations.

`MasterState` contract inherits from multiple contracts (most of the following contracts have the same inheritance model) related to the Master state.

The only publicly-accessible state-changing functions are for liquidations.

The contract is upgradeable (inherits from `UUPSUpgradeable`).

MiddleLayer

The contract is settled on each chain for communication between the Prime contracts and the message-passing architecture (eg. Axelar). The Axelar Route contract triggers actions on the MiddleLayer that delegates these actions to the relevant contracts.

The contract holds a mapping of authorized contracts and routes. Authorized contracts can call the `msend` function that forwards the message to the chosen route, and routes can call the `mreceive` function that forwards the message to Prime contracts based on the passed selector.

ECC

`ECC` (Error Correcting Contract) implements the logic for a store of data. It uses 8 bytes long blocks where are stored all the data needed for message passing. It implements its own logic for eliminating collisions in storage. Also, the contract allows pre-registration of messages (`preRegMsg`) that can be later processed (e.g., with `resendMessage`).

AxelarRoute

`AxelarRoute` can be one of many routes that can be used for communication between MiddleLayer and Axelar Gateway.

PToken

`PToken` contract is deployed on satellite chains on a per-supported asset basis. It allows users to add (`deposit`) and remove (`withdraw`) underlying assets as collateral.

The contract is upgradeable (inherits from `UUPSUpgradeable`).

LoanAsset

LoanAsset is a multi-chain ERC-20 token that is used for loans. Users can transfer tokens to another chain, and authorized addresses (Mint Authority) can mint tokens.

LoanAgent

`LoanAgent` contract is used for the management of loans and is supposed to be exactly one on each chain. `LoanAgent` allows users to `borrow`, `repayBorrow`, and `repayBorrowBehalf` of another borrower.

The contract is upgradeable (inherits from `UUPSUpgradeable`).

Treasury

The contract holds reserve tokens and allows arbitrageurs to mint/burn loan assets in exchange to trade assets. Its primary goal is to maintain the peg of loan assets.

CRM

`CRM` (Collateral Ratio Model) calculates how much the user can borrow against his deposits. Also, it utilizes a premium rate that is used for user incentivization. Loan market premium is calculated based on `ratioFloor` and `ratioCeiling` params.

The following image shows the loan market premium curve for `ratioFloor = 97%` and `ratioCeiling = 103%`.

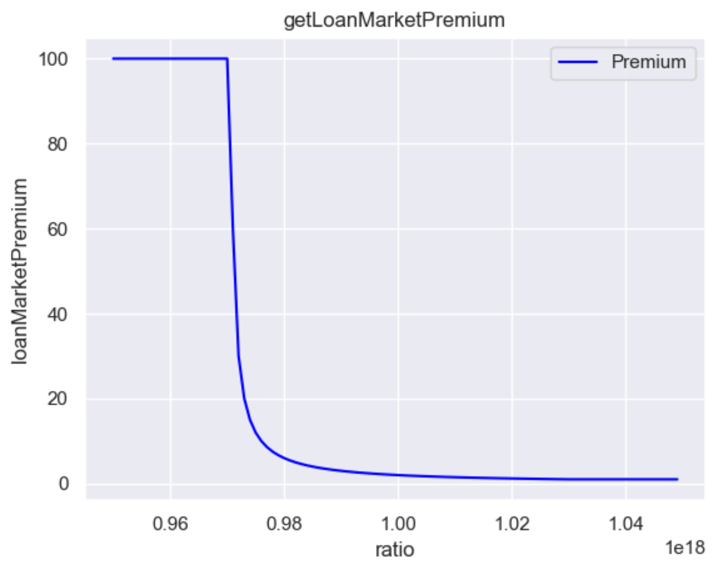

Figure 2. CRM.getLoanMarketPremium math model

IRM

`IRM` (Interest Rate Model) is used to increase or decrease the interest rate of various supported loan assets based on different factors. Its main goal should be to assist the stablecoins in maintaining the peg with the backing assets.

PrimeOracle

The contract responsible for accessing prices from various feeds (like `ChainlinkFeedGetter`).

SafeTransfers

The contract for safe transfers of tokens. It implements its own logic for validating returned value from token transfers into the contract (`_doTransferIn`) and out from the contract (`_doTransferOut`).

Actors

Admin

Each component of the protocol has its own Admin. The Admin is responsible for upgrading the contracts and setting the parameters, like adding new routes, markets or updating ratios, modifying supported loan/trade assets in `Treasury`, etc.

Middle Layer

Middle Layer is the only role that has access to modify `MasterState` in terms of deposits, withdrawals and borrows.

On satellite chains it has permissions to approve borrows in `LoanAgent` or mint from chain. In `PToken` contract the `MiddleLayer` can call `completeWithdraw` and `seize`.

Axelar Route

Axelar Routes are contracts used to pass messages from MiddleLayer to Axelar Gateway. Routes can access MiddleLayer's `mreceive` function.

Axelar Gateway

Axelar Gateway accepts messages from Axelar Routes and passes them through the Axelar message-passing architecture to other gateways on different chains.

Mint Authority

Mint Authority is the only role that can mint and burn loan assets.

ECC Authority

ECC Authority is the only role that can register messages or flag them as validated in ECC.

User

User can perform deposits, withdrawals, borrows, repays on satellite chains and liquidations, and accrual of interest on the master state.

5.2. Trust Model

The protocol highly relies on the administrators of the contracts. If any of the administrators is compromised, the protocol can be exploited critically in various ways. Also, the admin could set critical protocol parameters wrong and cause a lot of disastrous scenarios. Users have to trust the administrators to not abuse their power.

M1: USP can return different exchange rate

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	PriceOracle.sol	Type:	Logic error

Listing 1. Excerpt from /contracts/master/oracle//PrimeOracle.sol#L82-L93[PrimeOracle.getBorrowAssetExchangeRate]

```
        if (loanMarketUnderlying == uspAddress) {
            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);
        }
```

Description

When `uspAddress` is not set after deployment, then the variable is equal to zero-address and thus `getBorrowAssetExchangeRate`, can return different values than is expected because of USP-specific branching.

The `getUnderlyingPriceBorrow` is safe from this because it is checking for decimals (on zero-address).

Exploit scenario

Admin forgets to set USP address. As a result, `getBorrowAssetExchangeRate` returns a different value than is expected (also depending on the arguments - zero-address/USP as an underlying).

For example, the user calls `getBorrowAssetExchangeRate(someAsset, 1, ZERO_ADDRESS)`:

- zero-address belongs to a native token,
- so the call returns price for the native token,
- then `uspAddress` is set to USP address,
- and the next same call returns the asset ratio between `someAsset` and the native token.

Recommendation

Ensure that `uspAddress` is not set to zero-address. For example initialize it in the constructor with `address(1)`.

Solution (Revision 1.1)

The function `getUnderlyingPriceBorrow` is now reverting if `uspAddress` is set to zero-address.

Go back to Findings Summary

M2: Duplicate routes can cause loss of funds

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	MiddleLayer.sol	Type:	Logic error

Listing 2. Excerpt from /contracts/middleLayer/MiddleLayerAdmin.sol#L70-L74[MiddleLayerAdmin.addRoute]

```
    function addRoute(IRoute _newRoute) external onlyAdmin() {
        if(address(_newRoute) == address(0)) revert AddressExpected();
        routes.push(_newRoute);
        authRoutes[address(_newRoute)] = true;
    }
```

Listing 3. Excerpt from /contracts/middleLayer/MiddleLayer.sol#L69-L69[MiddleLayer.msend]

```
            uint256 hash = uint256(keccak256(abi.encodePacked(_params,
block.timestamp, _dstChainId)));
```

Description

It is possible to add multiple same routes via the `addRoute` function (see Listing 2). This behavior can cause there will be a bigger chance that the duplicated route will be chosen by the route picker (see Listing 3).

This issue becomes more several when the duplicated route is removed. The route is disabled from the mapping of the authorized routes,

```
authRoutes[address(_fallbackAddressToRemove)] = false;
```

however, the route is still in the route list (because of duplication). When is the `msend` function called, there is a chance (depending on the number of routes and duplications) that the disabled route will be chosen. This can cause a loss of funds.

Exploit scenario

Admin accidentally adds the same route again. Later he/she decides to remove the route and didn't notice the route was added twice. When the `msend` function is called, the disabled route is chosen and the passed funds are lost.

Recommendation

Prevent adding duplicated routes in the `addRoute` function.

Solution (Revision 1.1)

The `addRoute` function is modified to check if the route is already added. If so, the function will revert.

```
if (authRoutes[address(_newRoute)]) revert RouteExists();
```

Go back to Findings Summary

M3: Admin role can be renounced

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	**/*	Type:	Data validation

Listing 4. Excerpt from /contracts/ecc/ECC.sol#L81-L85[ECC.changeAdmin]

```
    function changeAdmin(
        address newAdmin
    ) external onlyAdmin() {
        admin = newAdmin;
    }
```

Description

The `changeAdmin` function lacks zero-address validation (see Listing 4). Due to that, the Admin role can be renounced by the current Admin.

Exploit scenario

The Admin accidentally calls `changeAdmin` with a zero-address. Then nobody will ever be able to use elevated privileges.

Recommendation

Add a zero-address check to prevent this if it is not intended. Otherwise, ignore this issue.

Solution (Revision 1.1)

The new contract `AdminControl.sol` has been added to the repository. It is a base contract that can be used to implement the Admin role. For changing the admin role, two-step process is used. First, the new admin is proposed.

Then, the proposed admin has to accept the role. This system prevents the accidental renouncement of the Admin role.

Go back to Findings Summary

M4: Two-phase Admin role transfer

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	`**/*`	Type:	Data validation

Description

Multiple contracts in the codebase use the `owner` pattern for access control and also allow ownership transfer.

However, neither of the transfer functions has a robust verification mechanism for the new proposed owner. If a wrong owner address is passed to them, neither can recover from the error.

Thus passing a wrong address can lead to irrecoverable mistakes.

Exploit scenario

The current owner Alice wants to transfer the ownership to Bob. Alice calls the `changeAdmin` function but supplies the wrong address by mistake. As a result, the ownership will be passed to the wrong address.

Recommendation

One of the common and safer approaches to ownership transfer is to use a two-step transfer process.

Suppose Alice wants to transfer the ownership to Bob. The two-step process would have the following steps: Alice proposes a new owner, namely Bob. This proposal is saved to a variable `candidate`. Bob, the candidate, calls the `acceptOwnership` function. The function verifies that the caller is the new proposed candidate, and if the verification passes, the function sets the caller as the new owner. If Alice proposes a wrong candidate, she can change it. However, it can happen, though with a very low probability that the wrong candidate is malicious (most often it would be a dead address). An authentication mechanism can be employed to prevent the malicious candidate from accepting the ownership.

Solution (Revision 1.1)

For changing the admin role, the two-step process is used. The logic is implemented in the new contract `AdminControl.sol`

Go back to Findings Summary

M5: The `setMidLayer` function has insufficient validation

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	ECC.sol	Type:	Data validation

Description

The `setMidLayer` function allows passing an arbitrary address.

Exploit scenario

By accident, an incorrect `newMiddleLayer` is passed to the function. Instead of reverting, the call succeeds.

Recommendation

Add more stringent data validation for `newMiddleLayer`. At the very least this would include a zero-address check. Ideally, we recommend defining a getter such as `contractId()` that would return a hash of an identifier unique to the (project, contract) tuple[1]. This will ensure the call reverts for most incorrectly passed values (see L1: Lack of project identifier for address validation for more information).

Solution (Revision 1.1)

The `setMidLayer` function now checks the unique ID identifier of `newMiddleLayer` by the added `isMiddleLayer` modifier.

M6: CRM missing validations

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	CRMAdmin.sol	Type:	Data validation

Description

`CRMStorage.sol` values `ratioCeiling` and `ratioFloor` are validated in the `CRM.sol` constructor but not in `CRMAdmin.sol` setters.

Listing 5. Excerpt from /contracts/master/crm//CRM.sol#L15-L18[CRMAdmin.constructor]

```
        if (
            ratioCeilingParam > 103e16 /* 103% */ ||
            ratioFloorParam < 97e16 /* 97% */
        ) revert ParamOutOfBounds();
```

Listing 6. Excerpt from /contracts/master/crm//CRMAdmin.sol#L8-L13[CRMAdmin.setRatioCeiling]

```
    function setRatioCeiling(
        uint256 ratio
    ) external onlyAdmin() returns (uint256) {
        ratioCeiling = ratio;
        return ratioCeiling;
    }
```

Listing 7. Excerpt from /contracts/master/crm//CRMAdmin.sol#L15-L20[CRMAdmin.setRatioFloor]

```
    function setRatioFloor(
        uint256 ratio
    ) external onlyAdmin() returns (uint256) {
            ratioFloor = ratio;
            return ratioFloor;
        }
```

Exploit scenario

Admin changes these values by intent (or by mistake), which leads to loan market premium manipulations and misbehaviors.

Recommendation

Add `ratioCeiling` and `ratioFloor` validations into setters in `CRMAdmin.sol`.

Solution ([Revision 1.1](#))

Two conditions have been added to `setRatioCeiling` and `setRatioFloor` functions in `CRMAdmin.sol` to validate the values.

[Go back to Findings Summary](#)

M7: IRM setters are not performing any kind of data validation

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	IRM.sol	Type:	Data validation

Description

The values for the IRM that are passed in the constructor are validated for non-zero values, however, the setters allow to set the values to zero.

Exploit scenario

Admin changes these values by intent (or by mistake) and critically affects the protocol.

Recommendation

Implement data validation for setters similarly as it is done in the constructor.

Solution (Revision 1.1)

The `IRM.sol` contract inherits the logic from the new contract `IRMAdmin.sol` that has been added to the repository. The new contract contains the logic for the setters with zero address checks.

Go back to Findings Summary

M8: Unsafe transfers

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Treasury.sol	Type:	Data validation

Listing 8. Excerpt from /contracts/satellite/treasury/Treasury.sol#L38-L38[Treasury.mintLoanAsset]

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

Listing 9. Excerpt from /contracts/satellite/treasury/Treasury.sol#L75-L75[Treasury.burnLoanAsset]

```
        else if (!_tradeAsset.transfer(msg.sender, tradeAmount)) revert
TransferFailed(msg.sender, address(this));
```

Description

The `Treasury.sol` inherits from `SafeTransfers` contract but does not use safe transfer methods on `ERC-20` assets.

Exploit scenario

A non-standard (or malicious) token is used in the contract. It causes successful transfers without transferring the amount (or any other unexpected behavior).

Recommendation

Use safe transfer functions from the `SafeTransfers` contract or use OpenZeppelin `SafeERC20` extension.

Solution (Revision 1.1)

The logic has been moved to the contract `TreasuryBase.sol` with the `SafeTransfers` inheritance, and the `_doTransferIn` and `_doTransferOut` functions are used.

Go back to Findings Summary

M9: Safe transfers are not checking for zero amounts

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	SafeTransfers.sol	Type:	Data validation

Description

The `doTransferOut` function does not check for zero amounts. This can lead to a transfer of zero tokens to a recipient address and not revert the transaction.

Exploit scenario

Bob performs a liquidation and sends some amount of tokens into MasterState. However, his reward is calculated as zero, and he loses his deposited tokens.

Recommendation

Add a requirement for a non-zero amount to the `_doTransferOut` function.

Solution (Revision 1.1)

The zero amount check with a revert has been added to the functions.

Go back to Findings Summary

M10: Duplicated balance values

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	Treasury.sol	Type:	Data validation

Listing 10. Excerpt from /contracts/satellite/treasury/Treasury.sol#L37-L37[Treasury.mintLoanAsset]

```
        assetReserves[tradeAsset] += tradeAmount;
```

Listing 11. Excerpt from /contracts/satellite/treasury/Treasury.sol#L71-L71[Treasury.burnLoanAsset]

```
        assetReserves[tradeAsset] = tradeAssetReserves - tradeAmount;
```

Description

`Treasury.sol` saves reserve assets' balances into the `assetReserves` mapping. This can cause inconsistencies between `assetReserves` and the real token balances in the contract in combination with unsafe transfers.

Exploit scenario

A malicious (non-standard) token performs a successful unsafe `transfer` without transferring tokens, but the balance in the `assetReserves` gets updated. Then `withdraw` and `burnLoanAsset` functions revert until these values match.

Listing 12. Excerpt from /contracts/satellite/treasury/TreasuryAdmin.sol#L22-L25[TreasuryAdmin.withdraw]

```
        if (assetAddress == address(0)) _assetReserves = address(
this).balance;
        else _assetReserves = ERC20(assetAddress).balanceOf(address(
this));

        if (assetReserves[assetAddress] > _assetReserves) revert
UnexpectedDelta();
```

Listing 13. Excerpt from /contracts/satellite/treasury/Treasury.sol#L65-L68[TreasuryAdmin.withdraw]

```
        if (tradeAsset == address(0)) tradeAssetReserves = address(
this).balance;
        else tradeAssetReserves = _tradeAsset.balanceOf(address(this));

        if (assetReserves[tradeAsset] > tradeAssetReserves) revert
UnexpectedDelta();
```

Recommendation

Use `SafeTransfers` to avoid balance miscalculations.

Solution ([Revision 1.1](#))

The functions from `SafeTransfers` are now used to transfer tokens.

[Go back to Findings Summary](#)

L1: Lack of project identifier for address validation

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	**/*	Type:	Data validation

Listing 14. Project Identifier

```
    bytes32 public constant CONTRACT_TYPE = keccak256("Prime - Middle
Layer");
```

Listing 15. Require statement for Data validation

```
    require(
        MiddleLayer(address_).CONTRACT_TYPE() == keccak256("Prime - Middle
Layer"),
        "Not a Middle Layer"
    );
```

Description

Currently, the contracts in constructors and setter functions are at most only checked against the zero address.

This approach can filter out the most basic mistakes, but it is not sufficient to ensure more deep address validation. Further validation can be done by using contract/project identifiers.

Such an identifier can be a constant string or a hash of a string (see Listing 14). Upon construction of a new contract that requires a Middle Layer address a check of the identifier would be done (see Listing 15). The same check can also be done anywhere else to ensure the correctness of the passed address.

Exploit scenario

A contract deployer passes a wrong address to a constructor of one of the Prime contracts. The address is not the zero address, but it is not a valid address of a Prime contract either. As a result, a contract is deployed with the wrong parameters.

Recommendation

It is recommended to use more stringent input data validation using the project-wide identifier - not only in the upgrade function but also in the constructors.

Such an approach might be not possible to implement when the contracts are circularly dependent on each other. Yet, this approach should be implemented where possible.

Solution ([Revision 1.1](#))

The issue has been fixed only for the `MiddleLayer`. We recommend applying unique ID validations also for other contracts. E.g., `MasterState` and `LoanAgent` in the `MiddleLayerAdmin`.

Client's comment: "Our deployment suite handles setting these addresses. Our solidity contracts will always be deployed and upgraded using our typescript infrastructure, and therefore this issue will not occur through our standardized deployment/operations procedure."

[Go back to Findings Summary](#)

L2: The `liquidateCalculateSeizeTokens` is not checking for a valid PToken address

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	MasterState.sol	Type:	Logic error

Description

Liquidations are not checking for a valid PToken address. This can lead to a revert after reaching Middle Layer since then will be called the `seize` function on the PToken address. If the `seize` function exists on the PToken address, it will be called, otherwise it will revert.

This is an uncontrolled call to an arbitrary address. Fortunately, the only way how to exploit that leads to loss of funds for the attacker. However, it is still a vulnerability.

Exploit scenario

Bob creates own PToken with an arbitrary `seize` function. Bob then calls the `liquidateBorrow` function with his PToken address as the parameter. The `seize` function will be called, and Bob can execute his code. Transaction will not revert and it will cause a loss of Bob's funds since the PToken will not transfer.

Recommendation

Add a check for a valid PToken address in the `liquidateCalculateSeizeTokens` function.

Solution (Revision 1.2)

Acknowledged. Client’s response:

” The auditors agree that this issue does not cause the protocol to lose funds, and can only result in the lost funds of a user attempting to pass a fraudulent PToken address. We do not think that an attacker losing their own funds is an issue for the protocol. ”

Go back to Findings Summary

W1: Treasury allows to receive native tokens without minting

Impact:	Warning	Likelihood:	N/A
Target:	Treasury.sol	Type:	Logic error

Description

The contract has an empty payable receive function. Due to this, a native token can be deposited without minting. This can be a problem in certain scenarios. For example, the token could not be withdrawn if there will be nothing to burn against.

Recommendation

Disable this feature if it is not intended for a production environment.

Solution (Revision 1.1)

The payable function `receive` has been removed.

Go back to Findings Summary

W2: Hardcoded decimals for native tokens

Impact:	Warning	Likelihood:	N/A
Target:	Treasury.sol	Type:	Arithmetics

Description

Decimals for native tokens are hardcoded in the Treasury contract. This could be potentially dangerous if any of the supported chains would differ from the hardcoded value.

Recommendation

Be aware of this issue if you will be adding some unconventional EVM chain to the protocol, or parametrize it.

Solution (Revision 1.1)

The native decimals are now set in the constructor of the `TreasuryStorage` contract.

Go back to Findings Summary

W3: Users can deposit but can not withdraw in a specific case

Impact:	Warning	Likelihood:	N/A
Target:	AxelarRoute.sol	Type:	Uninitialized values

Description

The `AxelarRoute` contract needs to set the `executor` to call the `execute` function (`onlyAX` modifier). The executor should be an Axelar Gateway address.

However, since the deposit process is not using the `execute` function on a source chain, it allows a successful deposit. However, users can not further withdraw their tokens if the `executor` is not set correctly.

This issue will not apply if the contract is on the same chain as the Master State.

Recommendation

Do not allow only partial functionality (if the user is not well informed about that). The `AxelarRoute` contract should not allow a deposit if the `executor` is not set correctly.

Solution (Revision 1.1)

The zero address check for the variable `executor` has been added to the constructor.

Go back to Findings Summary

W4: Inconsistent Master State values can break the calculations

Impact:	Warning	Likelihood:	N/A
Target:	MasterState.sol	Type:	Inconsistent state

Description

This issue presents an essential struggle in cross-chain projects, how to share critical values between different chains. The `MasterState` contract has `supportMarket` function that assigns values for a `PToken` instance, but these values can be set inconsistently against the real values that the `PToken` contract has on its chain. As a result, the calculations would be incorrect.

Recommendation

Values can be assigned using message-passing architecture or with a specific off-chain solution to ensure consistency (like deployment scripts).

Solution (Revision 1.1)

The client acknowledged the issue with the following comment: ”Using our deployment script, MasterState values are always verified and cannot be inconsistent.”

Go back to Findings Summary

W5: Missing `initializer` modifier on the constructor

Impact:	Warning	Likelihood:	N/A
Target:	MasterState.sol	Type:	Data validation

Description

Since the protocol is using a well-known upgradeability implementation (UUPSUpgradeable) the missing initializer can not affect the proxy contract. However, an attacker still can claim himself as the admin of the implementation contract and adjust the contract for his/her needs.

If the contract gets accidentally whitelisted or any other black swan event happens, the attacker can use the implementation contract as the potential attack vector for the protocol.

Recommendation

Add the `initializer` modifier on the constructor.

Solution (Revision 1.1)

The `initializer` modifier has been added to the constructor.

Go back to Findings Summary

W6: The `setBorrowRate` function does not emit events on different branching

Impact:	Warning	Likelihood:	N/A
Target:	IRM.sol	Type:	Events

Description

The `setBorrowRate` function emits the `SetBorrowRate` event only when `ratio > upperTargetRatio`, otherwise it does not emit any event, but the `borrowInterestRatePerBlock` variable is updated.

Recommendation

Emit events on every change of the `borrowInterestRatePerBlock` variable.

Solution (Revision 1.1)

The event is now correctly emitted at the end of the function.

Go back to Findings Summary

W7: Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Solution (Revision 1.2)

Acknowledged.

Go back to Findings Summary

W8: Lockfile overwriting

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Dependency management

Description

The `npm i` command overwrites the lockfile and that can cause undefined behavior.

Exploit scenario

A developer will go step by step with README in the repository to deploy its contracts. So, he/she will use `npm i` instead of `npm ci` (clean install) which will overwrite the lockfile. Contracts are deployed on an untested version and due to that contracts have different behavior than it's intended.

Recommendation

Use `npm ci` instead of `npm i` to install dependencies.

Solution (Revision 1.1)

README has been updated.

Go back to Findings Summary

I1: Inconsistent naming convention

Impact:	Info	Likelihood:	N/A
Target:	PTokenAdmin.sol	Type:	Code maturity

Description

The `isdeprecated` is not following the camel case naming convention.

Recommendation

Rename the function to `isDeprecated`.

Solution (Revision 1.2)

Acknowledged.

Go back to Findings Summary

I2: Misleading error for zero-address

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Custom errors

Description

The codebase uses the `AddressExpected()` error in cases where the zero-address is not allowed. This error is misleading since zero-address is still an address. Therefore, it does not reflect precisely what is happening.

Recommendation

The error should be renamed to something more clear like `ZeroAddressNotAllowed()`.

Solution (Revision 1.2)

Acknowledged.

Go back to Findings Summary

I3: Commented out code

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Code maturity

Description

The codebase contains commented-out code. This is a code smell and should be removed.

Recommendation

Remove all unnecessary code before use in a production environment.

Solution (Revision 1.2)

Acknowledged.

Go back to Findings Summary

I4: Inconsistent usage of (pre/post)incrementation

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Gas optimization

Description

The contract uses (pre/post)incrementation inconsistently in its for-loops. Pre-incrementation is the preferred way since it is cheaper for execution.

Recommendation

Replace post-incrementation with pre-incrementation in for-loops.

Solution (Revision 1.2)

Loops have been updated to pre-incrementation.

Go back to Findings Summary

I5: Unnecessary load

Impact:	Info	Likelihood:	N/A
Target:	ECC.sol	Type:	Gas optimization

Listing 16. Excerpt from /contracts/ecc/ECC.sol#L261-L267[ECC.resendMessage]

```
        if (!found) {
            bytes32 data;
            assembly {
                data := sload(ptr)
            }
            revert("rsm not found");
        }
```

Description

In the Listing 16 is unnecessary `sload` to the local variable since in the next line it will revert.

Recommendation

Remove the unnecessary code.

Solution (Revision 1.2)

Unused code has been removed.

Go back to Findings Summary

I6: LoanAgent code duplications

Impact:	Info	Likelihood:	N/A
Target:	LoanAgent.sol	Type:	Best practices

Description

The `LoanAgent` contract contains duplicated code in `repayBorrow` and `repayBorrwBehalf` functions.

Listing 17. Excerpt from /contracts/satellite/loanAgent//LoanAgent.sol#L61-L77[LoanAgent._sendRepay]

```
    function repayBorrow(
        address route,
        address loanMarketAsset,
        uint256 repayAmount
    ) external payable virtual override returns (uint256) {
        if (repayAmount == 0) revert ExpectedRepayAmount();
        if (loanMarketAsset == address(0)) revert AddressExpected();
        if (isFrozen[loanMarketAsset]) revert
MarketIsFrozen(loanMarketAsset);

        return _sendRepay(
            msg.sender,
            msg.sender,
            route,
            loanMarketAsset,
            repayAmount
        );
    }
```

Listing 18. Excerpt from /contracts/satellite/loanAgent//LoanAgent.sol#L85-L102[LoanAgent.repayBorrowBehalf]

```
    function repayBorrowBehalf(
        address borrower,
        address route,
        address loanMarketAsset,
        uint256 repayAmount
    ) external payable virtual override returns (uint256) {
        if (repayAmount == 0) revert ExpectedRepayAmount();
        if (loanMarketAsset == address(0)) revert AddressExpected();
        if (isFrozen[loanMarketAsset]) revert
MarketIsFrozen(loanMarketAsset);

        return _sendRepay(
            msg.sender,
            borrower,
            route,
            loanMarketAsset,
            repayAmount
        );
    }
```

Recommendation

Call the `repayBorrowBehalf` function from `repayBorrow` with `msg.sender` as a `borrower`.

```
function repayBorrow(
     address route,
     address loanMarketAsset,
     uint256 repayAmount
) external payable virtual override returns (uint256) {
    repayBorrowBehalf(msg.sender, route, loanMarketAsset, repayAmount)
}
```

Solution (Revision 1.2)

Code duplication has been resolved according to our recommendation.

Go back to Findings Summary

I7: ECC variables should be constants

Impact:	Info	Likelihood:	N/A
Target:	ECC.sol	Type:	Best practices

Description

State variables `maxSize`, `metadataSize`, and `usableSize` are assigned only in declarations. Should be contstants.

Listing 19. Excerpt from /contracts/ecc//ECC.sol#L64-L66[ECC.]

```
    uint256 internal maxSize = 8;
    uint256 internal metadataSize = 2;
    uint256 internal usableSize = 6;
```

Recommendation

Refactor these variables to constants and adjust the assembly code that uses it accordingly.

```
internal constant MAX_SIZE = 8;
uint256 internal constant METADATA_SIZE = 2;
uint256 internal constant USABLE_SIZE = 6;
```

Solution (Revision 1.2)

State variables have been transformed into constants, and the rest of the code has been updated accordingly.

Go back to Findings Summary

I8: Abstract contracts naming

Impact:	Info	Likelihood:	N/A
Target:	interfaces	Type:	Best practices

Description

`interfaces` folders sometimes contain abstract contracts with the `I` prefix names. Even these fake-interface abstract contracts extend other abstract contracts with state variables. This is very confusing and generally bad practice.

E.g. the `ILoanAgent` inherits from `LoanAgentStorage,` which contains state variables.

Listing 20. Excerpt from /contracts/satellite/loanAgent/interfaces/ILoanAgent.sol#L7-L7[ILoanAgent.]

```
abstract contract ILoanAgent is LoanAgentStorage {
```

Recommendation

Do not use `I` prefix for abstract contracts.

Solution ([Revision 1.2](#))

Acknowledged.

[Go back to Findings Summary](#)

I9: Documentation

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Best practices

Description

NatSpec documentation is missing in the majority of contracts (usually present only in interfaces). Some contracts with NatSpec documentation are missing param descriptions. E.g., `LoanAgent.sol` is missing `route` param description in all functions.

Recommendation

Cover all contracts and functions with NatSpec documentation. Missing or sporadic code documentation does not look professional in open-source projects.

Solution (Revision 1.2)

Acknowledged.

Go back to Findings Summary

[1] An example would be `keccak256("Prime - Middle Layer")`

6. Report revision 1.1

6.1. System Overview

The codebase has been updated as a response for some of the findings, and extended by the following contracts:

- `TreasuryBase.sol` - main logic has been moved from `Treasury.sol` to this contract.
- `TreasuryEvents.sol` - the new abstract contract.
- `AdminControl` - the new contract for managing the admin role.
- `IRMAdmin.sol` - setters have been moved from `IRM.sol` to this contract.

Trust model

The trust model has been updated in the following ways.

- Ownership transfer has been updated from single-step to two-step.
- Treasury contract now uses the contract ID validation.

7. Report revision 1.2

No significant changes in the code, only additional fixes according to Revision 1.1 feedback.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Prime: Protocol, January 3, 2023.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

ackee

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq