SMART CONTRACT AUDIT REPORT

for

Iron Lend

Prepared By: Yiqun Chen

PeckShield

August 31, 2021

Document Properties

Client	Iron Finance
Title	Smart Contract Audit Report For Iron Finance
Target	Iron Lend
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 31, 2021	Xuxian Jiang	Final Release
1.0-rc	August 21, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Iron Lend** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Iron Lend

The `Iron Lend` protocol is designed to enable a complete algorithmic money market protocol on `Polygon`. The protocol design is architected and forked based on `Compound` with a few minor changes. The protocol uses price oracles with 100% `ChainLink` price feeds instead of `Compound`'s open price feeds. It also directly mints reward token instead of transfers reward tokens to claimers. Finally, it removes some reward features related to `Contributors` which is not relevant to `Iron Finance`. Overall, it enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies.

The basic information of Iron Lend is as follows:

Table 1.1: Basic Information of Iron Lend

Item	Description
Name	Iron Finance
Website	https://iron.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 31, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/IronFinance/iron-lend.git (f9a6698)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Iron Lend protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	4	■ ■ ■ ■
Informational	1	■
Undetermined	1	■
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 4 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Iron Lend Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Adjusted blocksPerYear Constant in Interest Model	Business Logic	Fixed
PVE-002	Medium	Non ERC20-Compliance Of RToken	Coding Practices	Confirmed
PVE-003	Low	Possible Front-running For Unintended Payment In repayBorrowBehalf()	Time And State	Confirmed
PVE-004	Undetermined	Improved Reward Management in grantRewardInternal()	Business Logic	Fixed
PVE-005	Low	Consistency in IronController Setters	Coding Practice	Confirmed
PVE-006	Informational	Redundant State/Code Removal	Coding Practice	Confirmed
PVE-007	Low	Interface Inconsistency Between RErc20 And REther	Coding Practice	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Adjusted blocksPerYear Constant in Interest Model

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BaseJumpRateModelV2`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the `Iron Lend` protocol is heavily forked from `Compound`. Within the audited codebase, there is a contract `BaseJumpRateModelV2`, which, as the name indicates, is designed to provide a base interest rate model. While examining the specific interest rate implementation, we notice an issue that may affect the computed interest rate.

To elaborate, we show below the `BaseJumpRateModelV2` contract. It defines a constant state variable `blocksPerYear` the represents "the approximate number of blocks per year that is assumed by the interest rate model". It comes to our attention that the computation assumes the block time of 3 seconds per block, which should be 2 seconds per block on `Polygon`.

```
contract BaseJumpRateModelV2 {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock, uint
        jumpMultiplierPerBlock, uint kink);

    /**
     * @notice The address of the owner, i.e. the Timelock contract, which can update
         parameters directly
     */
    address public owner;

    /**
     * @notice The approximate number of blocks per year that is assumed by the interest
         rate model
```

```
     */
    uint public constant blocksPerYear = 10512000;

    /**
     * @notice The multiplier of utilization rate that gives the slope of the interest
         rate
     */
    uint public multiplierPerBlock;
    ...
    }
```

Listing 3.1: The `BaseJumpRateModelV2` Contract

Recommendation Revise the above constant state `blocksPerYear = 15768000` to apply the right block production time.

Status The issue has been fixed on the live contract with the $2.5s$ block time.

3.2 Non ERC20-Compliance Of RToken

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

Each asset supported by the `Iron Lend` protocol is integrated through a so-called `RToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `RToken` s, users can earn interest through the `RToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `RTokens` as collateral. There are currently two types of `RTokens`: `RErc20` and `REther`. In the following, we examine the ERC20 compliance of these `RTokens`.

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `RToken` contract. Specifically, the current `transfer()` function simply returns the related error

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example “Tether USD”	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example “USDT”. It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address’ balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

code if the sender does not have sufficient balance to spend. A similar issue is also present in the `transferFrom()` function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance. Also the emitted events when `RTokens` are minted should use `address(0)` instead of `address(this)`.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `RToken` implementation to ensure its ERC20-compliance.

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to minimize the difference from the original `Compound` and reduce the risk of introducing bugs as a result of changing the behavior.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.3 Possible Front-running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: RToken
- Category: Time and State [8]
- CWE subcategory: CWE-663 [4]

Description

The Iron Lend protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., mint()/redeem() and borrow()/repay(). In the following, we examine one specific functionality, i.e., repay().

To elaborate, we show below the core routine repayBorrowFresh() that actually implements the main logic behind the repay() routine. This routine allows for repaying partial or full current borrowing balance. It is interesting to note that the Iron Lend protocol supports the payment on behalf of another borrowing user (via repayBorrowBehalf()). And the repayBorrowFresh() routine supports the corner case when the given amount is larger than the current borrowing balance. In this corner case, the protocol assumes the intention for a full repayment.

```
function repayBorrowFresh(address payer, address borrower, uint repayAmount)
    internal returns (uint, uint) {
```

```
        /* Fail if repayBorrow not allowed */
        uint allowed = ironController.repayBorrowAllowed(address(this), payer, borrower,
             repayAmount);
        if (allowed != 0) {
            return (failOpaque(Error.IRON_CONTROLLER_REJECTION, FailureInfo.
                REPAY_BORROW_IRON_CONTROLLER_REJECTION, allowed), 0);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.
                REPAY_BORROW_FRESHNESS_CHECK), 0);
        }

        RepayBorrowLocalVars memory vars;

        /* We remember the original borrowerIndex for verification purposes */
        vars.borrowerIndex = accountBorrows[borrower].interestIndex;

        /* We fetch the amount the borrower owes, with accumulated interest */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return (failOpaque(Error.MATH_ERROR, FailureInfo.
                REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr))
                , 0);
        }

        /* If repayAmount == -1, repayAmount = accountBorrows */
        if (repayAmount == uint(-1)) {
            vars.repayAmount = vars.accountBorrows;
        } else {
            vars.repayAmount = repayAmount;
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We call doTransferIn for the payer and the repayAmount
         *  Note: The RToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the RToken holds an additional repayAmount of cash.
         *  doTransferIn reverts if anything goes wrong, since we can't be sure if side
              effects occurred.
         *   it returns the amount actually transferred, in case of a fee.
         */
        vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

        /*
         * We calculate the new borrower and total borrow balances, failing on underflow
              :
         *  accountBorrowsNew = accountBorrows - actualRepayAmount
```

```
         *  totalBorrowsNew = totalBorrows - actualRepayAmount
         */
        (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.
            actualRepayAmount);
        require(vars.mathErr == MathError.NO_ERROR, "
            REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");

        (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.
            actualRepayAmount);
        require(vars.mathErr == MathError.NO_ERROR, "
            REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a RepayBorrow event */
        emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew
            , vars.totalBorrowsNew);

        /* We call the defense hook */
        // unused function
        // ironController.repayBorrowVerify(address(this), payer, borrower, vars.
            actualRepayAmount, vars.borrowerIndex);

        return (uint(Error.NO_ERROR), vars.actualRepayAmount);
    }
```

Listing 3.2: `RToken::repayBorrowFresh()`

This is a reasonable assumption, but our analysis shows this assumption may be taken advantage of to launch a front-running `borrow()` operation, resulting in a higher borrowing balance for repayment. To avoid this situation, it is suggested to disallow the repayment amount of -1 to imply the full repayment. In fact, it is always suggested to use the exact payment amount in the `repayBorrowBehalf()` case.

Recommendation Revisit the generous assumption of using repayment amount of -1 as the indication of full repayment.

Status This issue has been confirmed. Considering the given amount is the choice from the repayer, the team decides to fix in the next upgrade.

3.4 Improved Reward Management in grantRewardInternal()

- ID: PVE-004
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: IronController
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

The Iron Lend has followed the same incentive mechanism as Compound to reward protocol users, either borrowers or lenders. However, one change made in Iron Lend is to directly mint reward token instead of transfers reward tokens to claimers.

To elaborate, we show below the grantRewardInternal() routine in IronController. This routine is designed to directly mint the reward amount to the claimer. As a security precaution, we would like to suggest to ensure the minted amount is within the intended range. This precaution may come handy to better regulate the total supply of reward tokens. Note this extra check is possible since both the reward rate and the elapsed time are readily available for reward calculation.

```
    function grantRewardInternal(address user, uint amount) internal returns (uint) {
        if (amount == 0) {
            return 0;
        }
        IRewardToken reward = IRewardToken(getRewardAddress());
        reward.mint(user, amount);
        return 0;
    }
```

Listing 3.3: IronController::grantRewardInternal()

Recommendation Add an extra layer of protection to ensure the reward tokens are minted according to the intended speed.

Status The issue has been fixed by having the logic inside the reward token contract to prevent excessive minting.

3.5 Consistency in IronController Setters

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: IronController
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The Iron Lend is heavily forked from Compound and shares the same key components, such as the protocol-wide controller. This controller acts as the gatekeeper to validate various operations, including mint(), redeem(), borrow(), repay(), liquidate(), and transfer(). While examining this controller, we notice one specific setter that can be improved.

To elaborate, we show below the _setCloseFactor() routine. This setter is designed to configure the closeFactor, a protocol-wide parameter used when a borrow position is liquidated. It comes to our attention this setter is different from others in the possibility of reverting the transaction when the caller is not the authorized admin. Note other setters have been designed to gracefully return a failure when the caller is not authorized. For consistency, we also suggest to gracefully return a failure as well instead of reverting the current transaction for unauthorized callers.

```
    /**
      * @notice Sets the closeFactor used when liquidating borrows
      * @dev Admin function to set closeFactor
      * @param newCloseFactorMantissa New close factor, scaled by 1e18
      * @return uint 0=success, otherwise a failure
      */
    function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
        // Check caller is admin
        require(msg.sender == admin, "only admin can set close factor");

        uint oldCloseFactorMantissa = closeFactorMantissa;
        closeFactorMantissa = newCloseFactorMantissa;
        emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

        return uint(Error.NO_ERROR);
    }
```

Listing 3.4: IronController::_setCloseFactor()

Recommendation Ensure the consistency in current setters by gracefully returning a failure information when an error occurs.

Status This issue has been confirmed. Considering that this is part of the original Compound code base, the team decides to leave it as is to minimize the difference from the original Compound and reduce the risk of introducing bugs as a result of changing the behavior.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The Iron Lend protocol makes good use of a number of reference contracts, such as ERC20, SafeBEP20 , SafeMath, and Address, to facilitate its code implementation and organization. For example, the IronController smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the RToken contract, there are a number of local variables that are defined, but not used. Examples include the err field in the defined MintLocalVars and RedeemLocalVars structures.

```
    struct MintLocalVars {
        Error err;
        MathError mathErr;
        uint exchangeRateMantissa;
        uint mintTokens;
        uint totalSupplyNew;
        uint accountTokensNew;
        uint actualMintAmount;
    }
```

Listing 3.5: RToken::MintLocalVars

In addition, the _acceptAdmin() routine in both IronDelegateController and RToken can be improved by removing the following redundant condition validation: msg.sender == address(0) (at lines 110 and 1133 respectively)

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been confirmed.

3.7 Interface Inconsistency Between RErc20 And REther

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.2, each asset supported by the `Iron Lend` protocol is integrated through a so-called `RToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. And `RTokens` are the primary means of interacting with the `Iron Lend` protocol when a user wants to `mint()`, `redeem()`, `borrow()`, `repay()`, `liquidate()`, or `transfer()`. Moreover, there are currently two types of `RTokens`: `RErc20` and `REther`. Both types expose the ERC20 interface and they wrap an underlying `ERC20` asset and `Ether/Matic`, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the `replayBorrow()` function as an example, the `RErc20` type returns an error code while the `REther` type simply reverts upon any failure. The similar inconsistency is also present in other routines, including `repayBorrowBehalf()`, `mint()`, and `liquidateBorrow()`.

```
    /**
     * @notice Sender repays their own borrow
     * @param repayAmount The amount to repay
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function repayBorrow(uint repayAmount) external returns (uint) {
        (uint err,) = repayBorrowInternal(repayAmount);
        return err;
    }
```

Listing 3.6: `RErc20::repayBorrow()`

```
    /**
     * @notice Sender repays their own borrow
     * @dev Reverts upon any failure
     */
    function repayBorrow() external payable {
        (uint err,) = repayBorrowInternal(msg.value);
        requireNoError(err, "repayBorrow failed");
    }
```

Listing 3.7: `REther::repayBorrow()`

It is also worth mentioning that the `RErc20` type supports `_addReserves` while the `REther` type does not.

Recommendation Ensure the consistency between these two types: `RErc20` and `REther`.

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to minimize the difference from the original `Compound` and reduce the risk of introducing bugs as a result of changing the behavior.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Iron Lend` protocol. The system expands `Iron Finance` by presenting a unique, robust offering as a decentralized money market protocol. The protocol design is architected and forked based on `Compound` with a few minor changes. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.