SMART CONTRACT AUDIT REPORT

for

RIFI Protocol

Prepared By: Patrick Lou

PeckShield

April 19, 2022

Document Properties

Client	Rikkei Finance
Title	Smart Contract Audit Report
Target	RIFI Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 19, 2022	Xuxian Jiang	Final Release
1.0-rc	September 30, 2021	Xuxian Jiang	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the RIFI lending protocol, 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 Rikkei

Rikkei Finance is a DeFi platform with a number of core features, including open lending, cross-chain support, NFTs collateralization, as well as P2P insurance. Specifically, the protocol adopts the decentralised technology to innovate on old ideas like money lending, credit, and even insurance, and enable cross-chain integration, meaning they can accept digital assets that operate on any blockchain network and render it at an equivalent rate, making them a real-time currency exchange network as well. The audited RIFI lending protocol is an algorithmic money market that is inspired from Compound with the planned deployment on Binance Smart Chain (BSC). The basic information of the Rikkei protocol is as follows:

Table 1.1: Basic Information of The Rikkei Protocol

Item	Description
Issuer	Rikkei Finance
Website	https://rikkei.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 19, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the `Rikkei Finance` protocol assumes a trusted oracle, which is not part of this audit.

- https://github.com/rikkei-finance/rifi-protocol.git (b33243f)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/rikkei-finance/rifi-protocol.git (e5d3877)

1.2 About PeckShield

PeckShield Inc. [15] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [14]:

- 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 list of check 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 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) [13], 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.

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 1.3: The Full List of Check Items

Category	Check Item
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

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 Logics	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 `RIFI lending` implementation. 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 logics, 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	2
Low	8
Informational	1
Total	11

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 that 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 2 medium-severity vulnerabilities, 8 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key RIFI Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-002	Low	Proper dsrPerBlock() Calculation	Business Logic	Resolved
PVE-003	Medium	Non ERC20-Compliance Of RToken	Coding Practices	Resolved
PVE-004	Low	Possible Front-running For Unintended Payment In repayBorrowBehalf()	Time And State	Confirmed
PVE-005	Low	Interface Inconsistency Between RBep20 And RBinance	Coding Practice	Resolved
PVE-006	Low	Improved Sanity Checks in System/-Function Arguments	Coding Practice	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Low	Improved Handling of Corner Cases in Proposal Submission	Business Logic	Resolved
PVE-009	Informational	Redundant State/Code Removal	Coding Practice	Resolved
PVE-010	Low	Proper Initialization of Cointroller	Business Logic	Resolved
PVE-011	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [12]
- CWE subcategory: CWE-663 [6]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `RToken` as an example, the `borrowFresh()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 786) start before effecting the update on internal states (lines 789 – 791), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
737     function borrowFresh(address payable borrower, uint borrowAmount) internal returns (
            uint) {
738         /* Fail if borrow not allowed */
739         uint allowed = cointroller.borrowAllowed(address(this), borrower, borrowAmount);
```

```
        if (allowed != 0) {
            return failOpaque(Error.COINTROLLER_REJECTION, FailureInfo.BORROW_COINTROLLER_REJECTION, allowed);
        }

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

        /* Fail gracefully if protocol has insufficient underlying cash */
        if (getCashPrior() < borrowAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.BORROW_CASH_NOT_AVAILABLE);
        }

        BorrowLocalVars memory vars;

        /*
         * We calculate the new borrower and total borrow balances, failing on overflow:
         *  accountBorrowsNew = accountBorrows + borrowAmount
         *  totalBorrowsNew = totalBorrows + borrowAmount
         */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

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

        /*
         * We invoke doTransferOut for the borrower and the borrowAmount.
         *  Note: The rToken must handle variations between BEP-20 and ETH underlying.
         *  On success, the rToken borrowAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side
```

```
             effects occurred.
         */
        doTransferOut(borrower, borrowAmount);

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

        /* We emit a Borrow event */
        emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew
            );

        /* We call the defense hook */
        // unused function
        // cointroller.borrowVerify(address(this), borrower, borrowAmount);

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

Listing 3.1: `RToken::borrowFresh()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`. The similar issue is also present in other functions, including `redeemFresh()` and `repayBorrowFresh()` in other contracts, and the adherence of the `checks-effects-interactions` best practice is strongly recommended. We highlight that the very same issue has been exploited in a recent `Cream` incident [1] and therefore deserves special attention.

From another perspective, the current mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the `Cointroller`-level. In addition, each individual function can be self-strengthened by following the `checks-effects-interactions` principle

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status The issue has been fixed by the following commit: `f4ef622`.

3.2 Proper dsrPerBlock() Calculation

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: DAIInterestRateModel
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

As mentioned earlier, the Rifi lending protocol is heavily forked from Compound by capitalizing the pooled funds for additional interest. Within the audited codebase, there is a contract DAIInterestRateModel , which, as the name indicates, is designed to provide DAI-related interest rate model. While examining the specific interest rate implementation, we notice a cross-chain issue that may affect the computed DAI Savings Rate (DSR).

To elaborate, we show below the dsrPerBlock() function. It computes the intended DAI "savings rate per block (as a percentage, and scaled by 1e18)". It comes to our attention that the computation assumes the block time of 15 seconds per block, which should be 3 seconds per block on Binance Smart Chain (BSC).

```
    /**
     * @notice Calculates the Dai savings rate per block
     * @return The Dai savings rate per block (as a percentage, and scaled by 1e18)
     */
    function dsrPerBlock() public view returns (uint) {
        return pot
            .dsr().sub(1e27)  // scaled 1e27 aka RAY, and includes an extra "ONE" before
                subraction
            .div(1e9) // descale to 1e18
    }
```

Listing 3.2: DAIInterestRateModel::dsrPerBlock()

Note another routine poke() within the same contract shares the same issue. Also, the BaseJumpRateModel contract and the LitePaperInterestRateModel contract implicitly assume the blocksPerYear to be 2102400, which is the case for the Ethereum deployment, but not the BSC deployment.

Recommendation Revise the above two functions (dsrPerBlock() and poke()) to apply the right block production time.

Status The issue has been fixed by the following commit: 395257f.

3.3 Non ERC20-Compliance Of RToken

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RToken`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [3]

Description

Each asset supported by the `Rifi lending` protocol is integrated through a so-called `RToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol.

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	✓

By minting `RTokens`, 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`: `RBep20` and `RBinance`. 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).

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()	✓

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 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.

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.

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	✓

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

Status The issue has been fixed by the following commit: `d56a00d`.

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

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `RToken`
- Category: Time and State [12]
- CWE subcategory: CWE-663 [6]

Description

The `Rifi lending` 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 `Rifi lending` 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 = cointroller.repayBorrowAllowed(address(this), payer, borrower,
            repayAmount);
        if (allowed != 0) {
            return (failOpaque(Error.COINTROLLER_REJECTION, FailureInfo.
                REPAY_BORROW_COINTROLLER_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 BEP-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
        // cointroller.repayBorrowVerify(address(this), payer, borrower, vars.actualRepayAmount, vars.borrowerIndex);

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

Listing 3.3: `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 leave it as is.

3.5 Interface Inconsistency Between RBep20 And RBinance

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.2, each asset supported by the `Rifi lending` 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 `Rifi lending` protocol when a user wants to `mint()`, `redeem()`, `borrow()`, `repay()`, `liquidate()`, or `transfer()`. Moreover, there are currently two types of `RTokens`: `RBep20` and `RBinance`. Both types expose the ERC20 interface and they wrap an underlying `BEP20` asset and `BNB`, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the `replayBorrow()` function as an example, the `RBep20` type returns an error code while the `RBinance` 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;
    }return err;
    }
```

Listing 3.4: RBep20::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 3.5: RBinance::repayBorrow()

It is also worth mentioning that the `RBep20` type supports `_addReserves` while the `RBinance` type does not.

Recommendation Ensure the consistency between these two types: `RBep20` and `RBinance`.

Status The issue has been fixed by the following commit: `d56a00d`.

3.6 Improved Sanity Checks in System/Function Arguments

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [3]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Rifi lending` protocol is no exception. Specifically, if we examine the `Cointroller` contract, it has defined a number of protocol-wide risk parameters, e.g., `liquidationIncentiveMantissa`, `collateralFactorMantissa` and `closeFactorMantissa`. In the following, we show an example routine that allows for their changes.

```
    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 3.6: `Cointroller::_setCloseFactor()`

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large `closeFactorMantissa` parameter will revert every liquidate operation.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status The issue has been confirmed and the team decides to exercise extra care in configuring these parameters with the further use of timelock to greatly reduce the risk.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [9]
- CWE subcategory: CWE-287 [4]

Description

In the `Rifi lending` protocol, there is a privileged `admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and incentive adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function _setVotingDelay(uint newVotingDelay) external {
        require(msg.sender == admin, "GovernorBravo::_setVotingDelay: admin only");
        require(newVotingDelay >= MIN_VOTING_DELAY && newVotingDelay <= MAX_VOTING_DELAY
            , "GovernorBravo::_setVotingDelay: invalid voting delay");
        uint oldVotingDelay = votingDelay;
        votingDelay = newVotingDelay;

        emit VotingDelaySet(oldVotingDelay,votingDelay);
    }

    /**
      * @notice Admin function for setting the voting period
      * @param newVotingPeriod new voting period, in blocks
      */
    function _setVotingPeriod(uint newVotingPeriod) external {
        require(msg.sender == admin, "GovernorBravo::_setVotingPeriod: admin only");
        require(newVotingPeriod >= MIN_VOTING_PERIOD && newVotingPeriod <=
            MAX_VOTING_PERIOD, "GovernorBravo::_setVotingPeriod: invalid voting period")
            ;
        uint oldVotingPeriod = votingPeriod;
        votingPeriod = newVotingPeriod;

        emit VotingPeriodSet(oldVotingPeriod, votingPeriod);
```

```
    }
```

Listing 3.7: `GovernorBravoDelegate::_setVotingDelay()/_setVotingPeriod()`

Note that if the privileged `admin` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig `admin` account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and the team plans to upgrade the admin with the use of a timelock. As the product and the community mature, the protocol will then move to a DAO-like governance model.

3.8 Improved Handling of Corner Cases in Proposal Submission

- ID: PVE-014
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GovernorAlpha`
- Category: Business Logic [11]
- CWE subcategory: CWE-837 [7]

Description

The `Rifi lending` protocol adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. In this section, we elaborate one corner case when a proposal is submitted regarding the proposer qualification.

Specifically, to be qualified as a proposer, the governance subsystem requires the proposer to obtain a sufficient number of votes, including from the proposer herself and other voters. The threshold is specified by `proposalThreshold()`. In `Rifi lending`, this number requires the votes of 100_000e18 (about 1% of `RIFI` token's total supply).

```
136   function propose(address[] memory targets, uint[] memory values, string[] memory signatures, bytes[] memory calldatas, string memory description) public returns (uint) {
137       require(rifi.getPriorVotes(msg.sender, sub256(block.number, 1)) > proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold");
138       require(targets.length == values.length && targets.length == signatures.length && targets.length == calldatas.length, "GovernorAlpha::propose: proposal function information arity mismatch");
139       require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
140       require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too many actions");

142       uint latestProposalId = latestProposalIds[msg.sender];
143       if (latestProposalId != 0) {
144         ProposalState proposersLatestProposalState = state(latestProposalId);
145         require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::propose: one live proposal per proposer, found an already active proposal");
146         require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha::propose: one live proposal per proposer, found an already pending proposal");
147       }
148       ...
149   }
```

Listing 3.8: GovernorAlpha::propose()

If we examine the `propose()` logic, when a proposal is being submitted, the governance verifies up-front the qualification of the proposer (line 137): `require(rifi.getPriorVotes(msg.sender, sub256(block.number, 1))> proposalThreshold())`. Note that the number of prior votes is strictly higher than `proposalThreshold()`.

However, if we check the proposal cancellation logic, i.e., the `cancel()` function, a proposal can be canceled (line 207) if the number of prior votes (before current block) is strictly smaller than `proposalThreshold()`. The corner case of having an exact number prior votes as the threshold, though unlikely, is largely unattended. It is suggested to accommodate this particular corner case as well.

```
202   function cancel(uint proposalId) public {
203       ProposalState state = state(proposalId);
204       require(state != ProposalState.Executed, "GovernorAlpha::cancel: cannot cancel executed proposal");

206       Proposal storage proposal = proposals[proposalId];
207       require(msg.sender == guardian   rifi.getPriorVotes(proposal.proposer, sub256(block.number, 1)) < proposalThreshold(), "GovernorAlpha::cancel: proposer above threshold");

209       proposal.canceled = true;
210       for (uint i = 0; i < proposal.targets.length; i++) {
211           timelock.cancelTransaction(proposal.targets[i], proposal.values[i], proposal
```

```
                .signatures[i], proposal.calldatas[i], proposal.eta);
        }

        emit ProposalCanceled(proposalId);
    }
```

Listing 3.9: GovernorAlpha::cancel()

Recommendation Accommodate the corner case by also allowing the proposal to be successfully submitted when the number of proposer's prior votes is exactly the same as the required threshold, i.e., `proposalThreshold()`.

Status The issue has been fixed by the following commit: `d56a00d`.

3.9 Redundant State/Code Removal

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-563 [5]

Description

The `Rifi lending` 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 `Cointroller` 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 3.10: RToken::MintLocalVars

Moreover, the `_acceptAdmin()` routine in both `Unitroller` and `RToken` can be improved by removing the following redundant condition validation: `msg.sender == address(0)` (at lines 110 and 1132 respectively)

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

Status The issue has been fixed by the following commit: `d56a00d`.

3.10 Proper Initialization of Cointroller

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Cointroller`
- Category: Business Logic [11]
- CWE subcategory: CWE-837 [7]

Description

As mentioned earlier, the `Rifi lending` protocol is heavily forked from `Compound` with an essential protocol-wide `Comptroller` or `Cointroller`. This contract mediates the access to various functionalities. While examining the contract logic, we notice its initialization can be improved.

To elaborate, we show below the `initialize()` function. It performs a basic logic in assigning the `rifiAddress`. However, it comes to our attention that it can be initialized by anyone – even though it can only be initialized once. To avoid unnecessary re-deployment from the mis-initialization, it is suggested to guard this function by ensuring the caller is from an authorized caller, say `admin`. Note this same issue is also applicable to the `SimplePriceOracle` contract

```
    function initialize(address rifi) public {
        require(rifiAddress == address(0), "RIFI address can only be set once");
        rifiAddress = rifi;
    }
```

Listing 3.11: Cointroller :: initialize ()

Recommendation Revise the above function `initialize()` to validate the authorized caller.

Status The issue has been fixed by the following commits: `d56a00d` and `e5d3877`.

3.11 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.12: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `drip()` routine in the `Reservoir` contract. If the `USDT` token is supported as `token_`, the unsafe version of `token_.transfer(target_, toDrip_)` (line 63) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
45  function drip() public returns (uint) {
46    // First, read storage into memory
47    EIP20Interface token_ = token;
48    uint reservoirBalance_ = token_.balanceOf(address(this)); // TODO: Verify this is a
        static call
49    uint dripRate_ = dripRate;
50    uint dripStart_ = dripStart;
51    uint dripped_ = dripped;
52    address target_ = target;
53    uint blockNumber_ = block.number;

55    // Next, calculate intermediate values
56    uint dripTotal_ = mul(dripRate_, blockNumber_ - dripStart_, "dripTotal overflow");
57    uint deltaDrip_ = sub(dripTotal_, dripped_, "deltaDrip underflow");
58    uint toDrip_ = min(reservoirBalance_, deltaDrip_);
59    uint drippedNext_ = add(dripped_, toDrip_, "tautological");

61    // Finally, write new `dripped` value and transfer tokens to target
62    dripped = drippedNext_;
63    token_.transfer(target_, toDrip_);

65    return toDrip_;
66  }
```

Listing 3.13: Reservoir :: drip ()

Note the same issue is also applicable to the `RDaiDelegate::doTransferIn()` function.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been confirmed. The team has considered the possible incompatibilities with other ERC-20 contracts and decided that the risk is minimal.

4 Conclusion

In this audit, we have analyzed the `RIFI lending` protocol design and implementation. The protocol is designed to be an algorithmic money market that is inspired from `Compound` with the planned deployment on `Binance Smart Chain (BSC)`. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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] Aislinn Keely. Cream Finance Exploited in $18.8 million Flash Loan Attack. https://www.theblockcrypto.com/linked/116055/creamfinance-exploited-in-18-8-million-flash-loan-attack.

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

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

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

[7] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

[9] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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

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

[16] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[17] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.