SMART CONTRACT AUDIT REPORT

for

ReactorFusion Protocol

Prepared By: Xiaomi Huang

PeckShield

May 31, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 31, 2023	Xuxian Jiang	Final Release
1.0-rc	May 29, 2023	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
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 ReactorFusion 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 ReactorFusion

ReactorFusion, a native lending and borrowing market on zkSync Era, is based on Compound Finance and offers unique bribe-reward tokenomics. By combining these powerful elements, ReactorFusion strives to provide the most advantageous incentives for money markets and maintain the deepest liquidity within the zkSync Era ecosystem. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The ReactorFusion Protocol

Item	Description
Name	ReactorFusion
Website	https://reactorfusion.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 31, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the protocol assumes a trusted price oracle with timely market price feeds for supported assets. And the oracle itself is not part of this audit.

- https://github.com/ReactorFusion/contracts.git (e3f4d30)

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

- https://github.com/ReactorFusion/contracts.git (51c6baf)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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

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

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) [10], 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 1.4: 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 `ReactorFusion` 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	1	■
Medium	1	■
Low	4	■■■■
Informational	0	
Total	6	

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 1 high-severity vulnerability, 1 medium-severity vulnerability, and 4 low-severity vulnerabilities.

Table 2.1: Key ReactorFusion Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect getSnapshots() Logic in CToken	Coding Practices	Resolved
PVE-002	High	Empty Market Avoidance With MINIMUM_LIQUIDITY Enforcement	Numeric Errors	Resolved
PVE-003	Low	Non ERC20-Compliance Of CToken	Coding Practices	Resolved
PVE-004	Low	Interface Inconsistency Between CErc20 And CEther	Coding Practice	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Potential Front-Running/MEV With Reduced Returns	Time And State	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 Incorrect getSnapshots() Logic in CToken

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

The `ReactorFusion` protocol is based on `Compound Finance` and offers unique bribe-reward tokenomics and related extensions. One specific extension is the exported interface, i.e., `getSnapshots()`, to return a snapshot of the account's balances and the cached exchange rate. Our analysis shows this interface has an issue in current implementation and needs to be revised.

To elaborate, we show below its implementation. This `getSnapshots()` routine has a rather straightforward logic in querying the account's balances and current exchange rate. However, there is an inconsistency in populating the return array. Specifically, the return array is defined as `uint256[2][]` (line 255), while the results are computed in an array `uint256[][2]` (lines 259 – 260). As a result, this routine has an unexpected inconsistency issue that needs to be resolved.

```
253     function getSnapshots(
254         address[] calldata accounts
255     ) external view returns (uint256[2][] memory) {
256         uint256[2][] memory ret = new uint256[2][](accounts.length);
257         for (uint256 i = 0; i < accounts.length; i++) {
258             address acc = accounts[i];
259             ret[i][0] = accountTokens[acc];
260             ret[i][1] = accountBorrows[acc].interestIndex == 0
261                 ? 0
262                 : ((accountBorrows[acc].principal * 1e18) /
263                     accountBorrows[acc].interestIndex);
264         }
265         return ret;
```

```
    }
```

Listing 3.1: `CToken::getSnapshots()`

Recommendation Revise the above `getSnapshots()` to resolve the inconsistency.

Status The issue has been fixed by this commit: `3572e35`.

3.2 Empty Market Avoidance With MINIMUM_LIQUIDITY Enforcement

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `CToken`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [3]

Description

The `ReactorFusion` 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()`. While reviewing the redeem logic, we notice the current implementation has a precision issue that has been reflected in a recent `HundredFinance` hack.

To elaborate, we show below the related `redeemFresh()` routine. As the name indicates, this routine is designed to redeems `CTokens` in exchange for the underlying asset. When the user indicates the underlying asset amount (via `redeemUnderlying()`), the respective `redeemTokens` is computed as `redeemTokens = div_(redeemAmountIn, exchangeRate)` (line 656). Unfortunately, the current approach may unintentionally introduce a precision issue by computing the `redeemTokens` amount against the protocol. Specifically, the resulting flooring-based division introduces a precision loss, which may be just a small number but plays a critical role when certain boundary conditions are met – as demonstrated in the recent `HundredFinance` hack: https://blog.hundred.finance/15-04-23-hundred-finance-hack-post-mortem-d895b618cf33.

```
626    function redeemFresh(
627        address payable redeemer,
628        uint redeemTokensIn,
629        uint redeemAmountIn
630    ) internal {
631        require(
632            redeemTokensIn == 0  redeemAmountIn == 0,
633            "one of redeemTokensIn or redeemAmountIn must be zero"
634        );
```

```
        /* exchangeRate = invoke Exchange Rate Stored() */
        Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal()});

        uint redeemTokens;
        uint redeemAmount;
        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
             * We calculate the exchange rate and the amount of underlying to be
                 redeemed:
             *  redeemTokens = redeemTokensIn
             *  redeemAmount = redeemTokensIn x exchangeRateCurrent
             */
            redeemTokens = redeemTokensIn;
            redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
            redeemTokens = div_(redeemAmountIn, exchangeRate);
            redeemAmount = redeemAmountIn;
        }

        /* Fail if redeem not allowed */
        uint allowed = comptroller.redeemAllowed(
            address(this),
            redeemer,
            redeemTokens
        );
        if (allowed != 0) {
            revert RedeemComptrollerRejection(allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            revert RedeemFreshnessCheck();
        }

        /* Fail gracefully if protocol has insufficient cash */
        if (getCashPrior() < redeemAmount) {
            revert RedeemTransferOutNotPossible();
        }

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

        /*
         * We write the previously calculated values into storage.
```

```
         *  Note: Avoid token reentrancy attacks by writing reduced supply before external transfer.
         */
        totalSupply = totalSupply - redeemTokens;
        accountTokens[redeemer] = accountTokens[redeemer] - redeemTokens;

        /*
         * We invoke doTransferOut for the redeemer and the redeemAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken has redeemAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
         */
        doTransferOut(redeemer, redeemAmount);

        dist.onAssetDecrease(bytes32("SUPPLY"), redeemer, redeemTokens);
        /* We emit a Transfer event, and a Redeem event */
        emit Transfer(redeemer, address(this), redeemTokens);
        comptroller.emitTransfer(
            redeemer,
            address(this),
            accountTokens[redeemer],
            0
        );
        comptroller.emitRedeem(redeemer, redeemAmount, redeemTokens);
        emit Redeem(redeemer, redeemAmount, redeemTokens);

        /* We call the defense hook */
        comptroller.redeemVerify(
            address(this),
            redeemer,
            redeemAmount,
            redeemTokens
        );
    }
```

Listing 3.2: `CToken::redeemFresh()`

Recommendation Properly revise the above routine to ensure the precision loss needs to be computed in favor of the protocol, instead of the user. In particular, we need to ensure that markets are never empty by minting small `CToken` balances at the time of market creation so that we can prevent the rounding error being used maliciously. A deposit as small as 1 wei is sufficient.

Status The issue has been resolved as the team confirms the following solution, i.e., Ensuring that markets are never empty by minting small `CToken` balances at the time of market creation prevents the rounding error being used maliciously. A deposit as small as 1 wei is sufficient. By having an initial deposit, the exploitable condition of an empty market can be avoided, making the attack infeasible.

3.3 Non ERC20-Compliance of CToken

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CToken
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

Each asset supported by the ReactorFusion protocol is integrated through a so-called CToken contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting CToken s, users can earn interest through the CToken's exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use CToken as collateral. There are currently two types of CToken: CErc20 and CEther. In the following, we examine the ERC20 compliance of these CTokens.

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	✓

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 `CToken` contract. Specifically, the current `mint()` function might emit the `Transfer` event with the contract itself as the source address. Note the ERC20 specification statest that "*A token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created.*" A similar issue is also present in the `transferFrom()` function.

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 `CToken` implementation to ensure its ERC20-compliance.

Status The issue has been fixed by this commit: `fcdf0bf`.

3.4 Interface Inconsistency Between CErc20 And CEther

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.2, each asset supported by the `ReactorFusion` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. And `CTokens` are the primary means of interacting with the `ReactorFusion` protocol when a user wants to `mint()`, `redeem()`, `borrow()`, `repay()`, `liquidate()`, or `transfer()`. Moreover, there are currently two types of `CTokens`: `CErc20` and `CEther`. Both types expose the ERC20 interface and they wrap an underlying `CErc20` asset and `Ether`, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the `replayBorrow()` function as an example, the `CErc20` type returns an error code while the `CEther` type does not have any return value. 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, or -1 for the full outstanding amount
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function repayBorrow(uint repayAmount) external override returns (uint) {
        repayBorrowInternal(repayAmount);
        return NO_ERROR;
    }

    /**
     * @notice Sender repays a borrow belonging to borrower
     * @param borrower the account with the debt being payed off
     * @param repayAmount The amount to repay, or -1 for the full outstanding amount
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function repayBorrowBehalf(
        address borrower,
        uint repayAmount
    ) external override returns (uint) {
        repayBorrowBehalfInternal(borrower, repayAmount);
        return NO_ERROR;
    }
```

Listing 3.3: `CErc20::repayBorrow()/repayBorrowBehalf()`

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

    /**
     * @notice Sender repays a borrow belonging to borrower
     * @dev Reverts upon any failure
     * @param borrower the account with the debt being payed off
     */
    function repayBorrowBehalf(address borrower) external payable {
        repayBorrowBehalfInternal(borrower, msg.value);
    }
```

Listing 3.4: `CEther::repayBorrow()/repayBorrowBehalf()`

Recommendation Ensure the consistency between these two types: `CErc20` and `CEther`.

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.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

In the `ReactorFusion` 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 marketing 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 _setMarketBorrowCaps(
        CToken[] calldata cTokens,
        uint[] calldata newBorrowCaps
    ) external {
        require(
            msg.sender == admin  msg.sender == borrowCapGuardian,
            "only admin or borrow cap guardian can set borrow caps"
        );

        uint numMarkets = cTokens.length;
        uint numBorrowCaps = newBorrowCaps.length;

        require(
            numMarkets != 0 && numMarkets == numBorrowCaps,
            "invalid input"
        );

        for (uint i = 0; i < numMarkets; i++) {
            borrowCaps[address(cTokens[i])] = newBorrowCaps[i];
            emit NewBorrowCap(cTokens[i], newBorrowCaps[i]);
        }
    }

    /**
     * @notice Admin function to change the Borrow Cap Guardian
     * @param newBorrowCapGuardian The address of the new Borrow Cap Guardian
```

```
     */
    function _setBorrowCapGuardian(address newBorrowCapGuardian) external {
        require(msg.sender == admin, "only admin can set borrow cap guardian");

        // Save current value for inclusion in log
        address oldBorrowCapGuardian = borrowCapGuardian;

        // Store borrowCapGuardian with value newBorrowCapGuardian
        borrowCapGuardian = newBorrowCapGuardian;

        // Emit NewBorrowCapGuardian(OldBorrowCapGuardian, NewBorrowCapGuardian)
        emit NewBorrowCapGuardian(oldBorrowCapGuardian, newBorrowCapGuardian);
    }

    /**
     * @notice Admin function to change the Pause Guardian
     * @param newPauseGuardian The address of the new Pause Guardian
     * @return uint 0=success, otherwise a failure. (See enum Error for details)
     */
    function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
        if (msg.sender != admin) {
            return
                fail(
                    Error.UNAUTHORIZED,
                    FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK
                );
        }

        // Save current value for inclusion in log
        address oldPauseGuardian = pauseGuardian;

        // Store pauseGuardian with value newPauseGuardian
        pauseGuardian = newPauseGuardian;

        // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
        emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);

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

Listing 3.5: Example `Setters` in the `Comptroller` Contract

If the privileged `admin` account is managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig 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 with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.6 Potential Front-Running/MEV With Reduced Returns

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: RewardDistributor
- Category: Time and State [8]
- CWE subcategory: CWE-682 [5]

Description

The ReactorFusion protocol provides a RewardDistributor contract that basically distributes available rewards. The reward distribution involves the toke swaps from USDC to WETH, and then to RF. With that, the protocol has provided a helper routine to facilitate the asset conversion: reap()

```
    function reap() public nonReentrant returns (uint256, uint256) {
        if (lastReap == block.timestamp) return (0, 0);
        require(msg.sender == address(underlying), "only underlying");
        // hardcoded to save gas
        IERC20 usdc = IERC20(0x3355df6D4c9C3035724Fd0e3914dE96A5a83aaf4);
        CToken ceth = CToken(0xC5db68F30D21cBe0C9Eac7BE5eA83468d69297e6);
        CToken cusdc = CToken(0x04e9Db37d8EA0760072e1aCE3F2A219988Fdac29);
        IPair ethrf = IPair(0x62eB02CB53673b5855f2C0Ea4B8fE198901F34Ac);
        IPair usdceth = IPair(0xcD52cbc975fbB802F82A1F92112b1250b5a997Df);
        uint256 vc_delta;
        uint256 vcBal = vc.balanceOf(address(this));
        uint256 rf_delta;

        if (block.timestamp - lastGaugeClaim >= duration) {
            rf_delta += swappedRF;
            swappedRF = 0;
            ceth.takeReserves();
            cusdc.takeReserves();
            uint256 wethTotal = 0;
            uint256 usdcbal = usdc.balanceOf(address(this));
            uint256 usdcWethOut = usdceth.getAmountOut(usdcbal, address(usdc));
            if (usdcWethOut > 0) {
                usdc.transfer(address(usdceth), usdcbal);
```

```
                usdceth.swap(0, usdcWethOut, address(ethrf), "");
                wethTotal += usdcWethOut;
            }
            ...
        }...
    }
```

Listing 3.6: `RewardDistributor::reap()`

To elaborate, we show above this helper routine. We notice the conversion is routed to `UniswapV2`-like pair in order to swap one asset to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status The issue has been fixed by this commit: `8e565a9`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `ReactorFusion` protocol, which is a native lending and borrowing market on `zkSync Era`. Based on `Compound Finance`, `ReactorFusion` offers unique bribe-reward tokenomics. By combining these powerful elements, `ReactorFusion` strives to provide the most advantageous incentives for money markets and maintain the deepest liquidity within the `zkSync Era` ecosystem. 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] 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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