SMART CONTRACT AUDIT REPORT

for

Chfry Protocol

Prepared By: Yiqun Chen

PeckShield

August 23, 2021

Document Properties

Client	Chfry
Title	Smart Contract Audit Report
Target	Chfry
Version	1.0
Author	Xuxian Jiang
Auditors	Jian Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 23, 2021	Xuxian Jiang	Final Release
1.0-rc2	August 15, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	June 20, 2021	Xuxian Jiang	Release Candidate #1
0.3	June 18, 2021	Xuxian Jiang	Additional Findings #2
0.2	June 15, 2021	Xuxian Jiang	Additional Findings #1
0.1	June 8, 2021	Xuxian Jiang	Initial Draft

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 design document and related smart contract source code of the Chfry 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 Chfry

The Chfry protocol is the DeFi protocol equivalent of fast food — its ecosystem powers tasty yields and satisfies cravings for multiple use cases. This is achieved by locking collateral within CHFRY — which supports the creation of synthetic yield-backed stablecoins, flash loans, leveraged yield farming and so on. The implementation enhances the initial Alchemix fork with flashloans support.

The basic information of the Chfry protocol is as follows:

Table 1.1: Basic Information of The Chfry Protocol

Item	Description
Name	Chfry
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 23, 2021

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

- https://github.com/chfry-finance/chfry-protocol.git (8fbbd47)

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

- https://github.com/chfry-finance/chfry-protocol.git (0cb8394)

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 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) [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.

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 Chfry 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	3	■■■
Low	4	■■■■
Informational	1	■
Total	8	

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 3 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Chfry Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-002	Low	Possible Initialization Prevention in TokenTimeRelease	Business Logic	Fixed
PVE-003	Medium	Lack Of Payment Source In executeTransaction()	Coding Practices	Fixed
PVE-004	Informational	Redundant State/Code Removal	Coding Practices	Confirmed
PVE-005	Low	Improved Logic in CDP::update()	Business Logic	Fixed
PVE-006	Low	Timely Reward Calculation in Oven::unstake()	Business Logic	Fixed
PVE-007	Medium	Accommodation of approve() Idiosyncrasies	Business Logic	Fixed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 [8]
- CWE subcategory: CWE-663 [3]

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` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `TokenTimeRelease` contract as an example, the `release()` 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`.

Apparently, the interaction with the external contract (line 105) starts before effecting the update on its internal state (line 106), 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.

```
99      /**
100      * @notice Transfers tokens held by timelock to beneficiary.
101      */
```

```
    function release() public virtual expectInitialized {
        uint256 amount = currentIncome();
        if (amount > 0) {
            token().safeTransfer(beneficiary(), amount);
            _releaseAmount = _releaseAmount.add(amount);
        }
    }
```

Listing 3.1: `TokenTimeRelease::release()`

In the meantime, we should mention that 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`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`.

Note similar issues exist in other functions, including `Oven::stake()` and `CheeseFactory::poolMint()`. The adherence of `checks-effects-interactions` best practice is strongly recommended.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `fe64707`.

3.2 Possible Initialization Prevention in TokenTimeRelease

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TokenTimeRelease`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Chfry` protocol has the flexible support of vesting schedules. It allows a beneficiary to extract the tokens after a given release time. The vesting schedule is implemented in `TokenTimeRelease`, which is managed by `TokenTimeLockManager`.

To elaborate, we show below the `initialize()` function from the `TokenTimeRelease` contract. As the name indicates, the `initialize()` function signals the kick-off of the intended vesting schedule. It comes to our attention it enforces the exact balance check `require(IERC20(_token).balanceOf(address(this))== _releaseTotalAmount)` (line 40), which is fragile as a small amount of balance may be forcibly transferred to the vesting contract even before it is created! As a result, a denial-of-service situation may be introduced to block the initialization of the intended vesting schedule.

```
    function initialize() external {
        require(!initialized, "already initialized");
        initialized = true;
        require(IERC20(_token).balanceOf(address(this)) == _releaseTotalAmount,"!balance
            ");
    }
```

Listing 3.2: `TokenTimeRelease::initialize()`

Recommendation Instead of strictly enforcing the equality in the above `initialize()` function, we suggest to revise it as follows:

```
    function initialize() external {
        require(!initialized, "already initialized");
        initialized = true;
        require(IERC20(_token).balanceOf(address(this)) >= _releaseTotalAmount,"!balance
            ");
    }
```

Listing 3.3: Revised `TokenTimeRelease::initialize()`

Status The issue has been fixed by this commit: `bbf8b19`.

3.3 Lack Of Payment Source In executeTransaction()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MultiSigWallet`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

The `Chfry` protocol has a core `MultiSigWallet` contract that contains a rather standard multi-signature implementation. The multi-signature support implements the well-established APIs, including `submitTransaction()`, `confirmTransaction()`, `revokeConfirmation()`, and `executeTransaction()`. While examining the actual transaction execution, we notice the invoked `external_call()` allows for the transfer of native tokens, e.g., `Ether`.

To elaborate, we show below the `executeTransaction()` routine as well as the helper `external_call()` routine. It comes to our attention that the `executeTransaction()` can be invoked when the transaction is being confirmed via `confirmTransaction()`. However, this `confirmTransaction()` function does not have the `payable` modifier, which makes the `external_call()` with native tokens inconvenient. For gas efficiency by avoiding explicit calls to deposit native tokens, it is suggested to make `executeTransaction()` callers `payable`, including `confirmTransaction()`, and `submitTransaction()`.

```
    /// @dev Allows anyone to execute a confirmed transaction.
    /// @param transactionId Transaction ID.
    function executeTransaction(uint256 transactionId)
        public
        virtual
        ownerExists(msg.sender)
        confirmed(transactionId, msg.sender)
        notExecuted(transactionId)
    {
        if (isConfirmed(transactionId)) {
            Transaction storage txn = transactions[transactionId];
            txn.executed = true;
            if (
                external_call(
                    txn.destination,
                    txn.value,
                    txn.data.length,
                    txn.data
                )
            ) emit Execution(transactionId);
            else {
                emit ExecutionFailure(transactionId);
                txn.executed = false;
            }
        }
    }

    // call has been separated into its own function in order to take advantage
    // of the Solidity's code generator to produce a loop that copies tx.data into
        memory.
    function external_call(
        address destination,
        uint256 value,
        uint256 dataLength,
        bytes memory data
    ) internal returns (bool) {
        bool result;
        assembly {
            let x := mload(0x40) // "Allocate" memory for output (0x40 is where "free
                memory" pointer is stored by convention)
            let d := add(data, 32) // First 32 bytes are the padded length of data, so
                exclude that
            result := call(
                sub(gas(), 34710), // 34710 is the value that solidity is currently
                    emitting
                // It includes callGas (700) + callVeryLow (3, to pay for SUB) +
                    callValueTransferGas (9000) +
                // callNewAccountGas (25000, in case the destination address does not
                    exist and needs creating)
                destination,
                value,
                d,
```

```
                dataLength, // Size of the input (in bytes) - this is what fixes the
                    padding problem
                x,
                0 // Output is ignored, therefore the output size is zero
            )
        }
        return result;
    }
```

Listing 3.4: `MultiSigWallet::executeTransaction()`

Recommendation Add the `payable` modifier to the above functions: `confirmTransaction()`, `submitTransaction()`, and `executeTransaction()`.

Status The issue has been fixed by this commit: `fd1ba46`.

3.4 Redundant State/Code Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CheeseToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

The `Chfry` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `Fryer` 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 `CheeseToken` contract, it is inherited form `ERC20` and `UpgradableProduct`. It turns out that the `UpgradableProduct` functionality is not used throughout the contract.

```
contract CheeseToken is ERC20, UpgradableProduct {
    using SafeMath for uint256;

    mapping(address => bool) public whiteList;

    constructor(string memory _symbol, string memory _name)
        public
        ERC20(_name, _symbol)
    {}
    ...
```

```
}
```

Listing 3.5: CheeseToken.sol

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

Status The issue has been confirmed.

3.5 Improved Logic in CDP::update()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `CDP`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Chfry` protocol, there is a one-to-one mapping between a `user` and the associated `CDP`. To properly manage the user's `CDP`, the protocol provides a library which includes the core CDP data struct and associated functions. While examining the current `CDP` functions, we notice the `update()` routine can be improved.

To elaborate, we show below the `update()` routine. It implements a rather straightforward logic in updating the `CDP`'s credit and debit amounts. However, in the case of the following `else`-branch, the `totalCredit` can be zeroed out just like the `totalDebt` case in the `then`-branch (line 38).

```
31     function update(Data storage _self, Context storage _ctx) internal {
32         uint256 _earnedYield = _self.getEarnedYield(_ctx);
33         if (_earnedYield > _self.totalDebt) {
34             uint256 _currentTotalDebt = _self.totalDebt;
35             _self.totalDebt = 0;
36             _self.totalCredit = _earnedYield.sub(_currentTotalDebt);
37         } else {
38             _self.totalDebt = _self.totalDebt.sub(_earnedYield);
39         }
40         _self.lastAccumulatedYieldWeight = _ctx.accumulatedYieldWeight;
41     }
```

Listing 3.6: `CDP::update()`

Recommendation Properly revise the above routine to update the `totalCredit` field. An example revision is shown below:

```
    function update(Data storage _self, Context storage _ctx) internal {
        uint256 _earnedYield = _self.getEarnedYield(_ctx);
        if (_earnedYield > _self.totalDebt) {
            uint256 _currentTotalDebt = _self.totalDebt;
            _self.totalDebt = 0;
            _self.totalCredit = _earnedYield.sub(_currentTotalDebt);
        } else {
            _self.totalCredit = 0;
            _self.totalDebt = _self.totalDebt.sub(_earnedYield);
        }
        _self.lastAccumulatedYieldWeight = _ctx.accumulatedYieldWeight;
    }
```

Listing 3.7: Revised `CDP::update()`

Status The issue has been fixed by this commit: `9303516`.

3.6 Timely Reward Calculation in Oven::unstake()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Oven`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

At the core of the `Chfry` protocol is the `Oven` contract that collects yield from the harvested gains. Users can stake `Fries` in the contract and their tokens will be converted into the base asset over time as the yield flows in. When users go to `Oven` and claim their converted tokens, an equal amount of `Fries` tokens will be burned.

```
    ///@dev Withdraws staked friesTokens from the exchange
    ///
    /// This function reverts if you try to draw more tokens than you deposited
    ///
    ///@param amount the amount of friesTokens to unstake
    function unstake(uint256 amount) public updateAccount(msg.sender) {
        // by calling this function before transmuting you forfeit your gained allocation
        address sender = msg.sender;
        require(
            depositedFriesTokens[sender] >= amount,
            "unstake amount exceeds deposited amount"
        );
        depositedFriesTokens[sender] = depositedFriesTokens[sender].sub(amount);
        totalSupplyFriesTokens = totalSupplyFriesTokens.sub(amount);
```

```
        friesToken.safeTransfer(sender, amount);
    }
```

Listing 3.8: `Oven::unstake()`

To elaborate, we show above the related `unstake()` routine. It comes to our attention that the current `unstake()` logic can be improved to timely call `runPhasedDistribution()` to run the phased distribution of the buffered funds – just like the `stake()` counterpart.

Recommendation Revise the `unstake()` routine to timely run the phased distribution of the buffered funds by calling `runPhasedDistribution()`.

Status The issue has been fixed by this commit: `73287fc`.

3.7 Accommodation of approve() Idiosyncrasies

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `FlashBorrower`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

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 this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
```

```
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.9: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. An example is shown below. It is in the `approveRepayment()` routine that is designed to specify the spending allowance. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function approveRepayment(address token, uint256 amount) public {
        uint256 _allowance =
            IERC20(token).allowance(address(this), address(lender));
        uint256 _fee = lender.flashFee(token, amount);
        uint256 _repayment = amount + _fee;
        IERC20(token).approve(address(lender), _allowance + _repayment);
    }
```

Listing 3.10: `FlashBorrower::approveRepayment()`

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status The issue has been fixed by this commit: `78043d2`.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `Chfry` protocol, the `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., adding new `vaults`, and configuring protocol parameters). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the related components. In the following, we show representative privileged operations in the protocol's core `Fryer` contract.

```
    function setOven(address _oven) external requireImpl {
        require(
            _oven != fryerConfig.ZERO_ADDRESS(),
            "oven address cannot be 0x0."
        );
        oven = _oven;
        emit OvenUpdated(_oven);
    }

    function setConfig(address _config) external requireImpl {
        require(
            _config != fryerConfig.ZERO_ADDRESS(),
            "config address cannot be 0x0."
        );
        fryerConfig = IFryerConfig(_config);
        _ctx.fryerConfig = fryerConfig;
        emit ConfigUpdated(_config);
    }

    function setFlushActivator(uint256 _flushActivator) external requireImpl {
        flushActivator = _flushActivator;
    }

    function setRewards(address _rewards) external requireImpl {
        require(
            _rewards != fryerConfig.ZERO_ADDRESS(),
            "rewards address cannot be 0x0."
        );
        rewards = _rewards;
        emit RewardsUpdated(_rewards);
    }

    function setOracleAddress(address Oracle, uint256 peg)
        external
        requireImpl
    {
        _linkGasOracle = Oracle;
        pegMinimum = peg;
    }
```

Listing 3.11: A number of representative `setters` in `Fryer`

We emphasize that the privilege assignment is necessary and consistent with the token design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that this privileged account will be managed by a multi-sig account. Note that a compromised `owner` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `Chfry` protocol.

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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Chfry` protocol. The audited system presents a unique addition to current DeFi offerings in maximizing yields for users. Developed on top of `Alchemix`, the `Chfry` protocol supports the creation of synthetic yield-backed stablecoins, flash loans, leveraged yield farming and so on. The current code base is clearly 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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