SMART CONTRACT AUDIT REPORT

for

Fountain Protocol

Prepared By: Patrick Liu

PeckShield

February 24, 2022

Document Properties

Client	Fountain Protocol
Title	Smart Contract Audit Report
Target	Fountain Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 24, 2022	Luck Hu	Final Release
1.0-rc	February 23, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Patrick Liu
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Fountain 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 ERC20-compliance, security or performance. This document outlines our audit results.

1.1 About Fountain Protocol

The Fountain protocol is the first cross-chain lending platform powered by Oasis. The protocol enables users to experience high capital efficiency one-stop management of DeFi assets. Taking advantage of the extremely efficient and low-cost Oasis network, Fountain establishes a multi-revenue protocol with a fund pool as the core and enables multiple application scenarios. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the Fountain Protocol

Item	Description
Name	Fountain Protocol
Website	https://ftp.cash/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 24, 2022

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

- https://github.com/dev-fountain/fountain-protocol.git (cc16318)

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

- https://github.com/dev-fountain/fountain-protocol.git (6986267)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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) [11], 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 `Fountain` protocol 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	6	■■■■■■
Informational	0	
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 2 medium-severity vulnerabilities, and 6 low-severity vulnerabilities.

Table 2.1: Key Fountain Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic Of LP-Farm::claimRewards()	Business Logic	Fixed
PVE-002	Medium	Non ERC20-Compliance Of CToken	Coding Practices	Fixed
PVE-003	Low	Possible Front-Running For Overpay In repayBorrowBehalf()	Time And State	Fixed
PVE-004	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-005	Low	Interface Inconsistency Between CErc20 And CEther	Coding Practice	Confirmed
PVE-006	Low	Timely _updateAllPools() During Pool Weight Changes	Business Logic	Fixed
PVE-007	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-008	Low	Proper dsrPerBlock() Calculation	Business Logic	Fixed

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 Improved Logic Of LPFarm::claimRewards()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LPFarm`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

One essential contract in the `Fountain` protocol is `LPFarm` that is designed to provide a farming solution. Users could deposit their LP tokens in `LPFarm` to earn rewards. While examining the logic to claim rewards, we notice the current implementation can be improved.

To elaborate, we show below the related `claimRewards()` function that is invoked when an user intends to claim rewards. This function calculates all the pending rewards of the user, and transfers the pending rewards to the user. However, there is a validation check in the `claimRewards()` function (line 178) which will stop the claiming when the protocol doesn't have sufficient balance for the pending rewards. As a result, the user can't receive any reward. In this corner case, it is expected that the user could claim the protocol's available balance. In other words, the `if`-condition of `pending > 0 && pending <= balance` should be revised to be `pending > 0`!

```
164     function claimRewards(address _account) external nonReentrant {
165         uint pending;
166         for(uint256 i = 0; i < poolInfo.length; i++){
167             PoolInfo storage pool = poolInfo[i];
168             UserInfo storage user = userInfo[i][_account];
169             updatePool(i);
170             if (user.amount > 0) {
171                 uint256 reward = user.amount * pool.accRewardsPerShare / PRECISION -
                        user.rewardDebt;
172                 pending += reward;
173                 emit ClaimRewards(_account, i, reward);
174             }
```

```
            user.rewardDebt = user.amount * pool.accRewardsPerShare / PRECISION;
        }
        uint256 balance = IERC20(rewardToken).balanceOf(address(this));
        if(pending > 0 && pending <= balance) {
            safeRewardsTransfer(_account, pending);
        }
    }
```

Listing 3.1: `LPFarm.sol::claimRewards()`

```
function safeRewardsTransfer(address to, uint amount) internal {
    uint rewardTokenBalance = IERC20(rewardToken).balanceOf(address(this));
    if(amount > rewardTokenBalance) {
        IERC20(rewardToken).safeTransfer(to, rewardTokenBalance);
    } else {
        IERC20(rewardToken).safeTransfer(to, amount);
    }
}
```

Listing 3.2: `LPFarm.sol::safeRewardsTransfer()`

Recommendation Revise the reward-claiming logic by adjusting the above-mentioned `if`-condition.

Status This issue has been fixed in the following commit: `94d3017`.

3.2 Non ERC20-Compliance Of CToken

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CToken`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [3]

Description

Each asset supported by the `Fountain` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `CTokens`, 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 `CTokens` as collateral. There are currently two types of `CTokens`: `CErc20` and `CEther`. In the following, we examine the ERC20 compliance of these `CTokens`.

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

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	✓

token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `CToken` 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 sufficient balance or the message sender does not have 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.

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

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

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

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

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

Status This issue has been fixed in the following commit: `ddcbffb`.

3.3 Possible Front-Running For Overpay In repayBorrowBehalf()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `CToken`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

As mentioned earlier, the `Fountain` 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 `Fountain` 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 = comptroller.repayBorrowAllowed(address(this), payer, borrower,
            repayAmount);
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockTimestamp != getBlockTimestamp()) {
            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 cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken 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
        // comptroller.repayBorrowVerify(address(this), payer, borrower, vars.
            actualRepayAmount, vars.borrowerIndex);

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

Listing 3.3: `CToken::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 fixed in the following commit by disallowing the repayment amount of -1 to imply the full repayment when the `payer` is not the `borrower`: `fe114dc`.

3.4 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CToken, Stake, LPFarm
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

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` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `CToken` 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 790) starts before effecting the update on internal states (lines 793 – 795), 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.

```
741 function borrowFresh(address payable borrower, uint borrowAmount) internal returns (
     uint) {
742     /* Fail if borrow not allowed */
743     uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
744     if (allowed != 0) {
745         return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                BORROW_COMPTROLLER_REJECTION, allowed);
746     }
747 
748     /* Verify market's block number equals current block number */
749     if (accrualBlockTimestamp != getBlockTimestamp()) {
750         return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
751     }
752 
753     /* Fail gracefully if protocol has insufficient underlying cash */
754     if (getCashPrior() < borrowAmount) {
755         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 cToken must handle variations between ERC-20 and ETH underlying.
     *  On success, the cToken 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
    // comptroller.borrowVerify(address(this), borrower, borrowAmount);

```

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

Listing 3.4: CToken::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 CToken, stake() and withdraw() in Stake contract, deposit(),depositBehalf() and withdraw() in LPFarm contract. 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 re-entrancy protection in CToken can be strengthened by elevating the re-entrancy protection at the Comptroller -level. In addition, each individual function can be self-strengthened by following the checks-effects -interactions principle

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

Status This issue has been fixed in the following commit: 94d3017.

3.5 Interface Inconsistency Between CErc20 And CEther

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CErc20, CEther
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.2, each asset supported by the Fountain 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 Fountain 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 ERC20 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 simply reverts upon any failure. The similar inconsistency is also present in other routines, including `repayBorrowBehalf()`, `mint()`, and `liquidateBorrow()`.

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

Listing 3.5: `CErc20::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 3.6: `CEther::repayBorrow()`

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.

3.6 Timely _updateAllPools() During Pool Weight Changes

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: LPFarm
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The Fountain protocol provides an incentive mechanism that rewards the staking of supported assets with the governance token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via add() and the weights of supported pools can be adjusted via set(). When analyzing the pool weight update routine set(), we notice the need of timely invoking _updateAllPools() to update the reward distribution before the new pool weight becomes effective.

```
    function set(uint _pid, uint _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            _updateAllPools();
        }
        uint prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        if (prevAllocPoint != _allocPoint) {
            updateStakingPool();
        }
    }
```

Listing 3.7: LPFarm::set()

If the call to _updateAllPools() is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the onlyOwner modifier), which greatly alleviates the concern.

Recommendation Timely invoke _updateAllPools() when any pool's weight has been updated. In fact, the third parameter (_withUpdate) to the set() routine can be simply ignored or removed.

```
    function set(uint _pid, uint _allocPoint, bool _withUpdate) public onlyOwner {
        _updateAllPools();
        uint prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
```

```
        if (prevAllocPoint != _allocPoint) {
            updateStakingPool();
        }
    }
```

Listing 3.8: Revised LPFarm::set()

Status This issue has been fixed in the following commit: `f7a3fc9`.

3.7 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [4]

Description

In the `Fountain` protocol, there exist certain privileged accounts that play critical roles in governing and regulating the system-wide operations. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

Firstly, the privileged functions in the `LPFarm` contract allow for the the `owner` to add a new token pool, update a given pool's reward allocation point, set reward amount per second and set the reward multiplier.

```
    function updateMultiplier(uint multiplierNumber) public onlyOwner {
        _updateAllPools();
        bounsMultiplier = multiplierNumber;
    }
    function add(uint _allocPoint, address _lpToken, bool _withUpdate) public onlyOwner
        {
        require(!tokenAddedList[_lpToken], "token exists");
        if (_withUpdate) {
            _updateAllPools();
        }
        uint lastRewardTime = getBlockTimestamp() > startTime ? getBlockTimestamp() :
            startTime;
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardTime: lastRewardTime,
            accRewardsPerShare: 0
        }));
        tokenAddedList[_lpToken] = true;
        updateStakingPool();
```

```
    }
    function set(uint _pid, uint _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            _updateAllPools();
        }
        uint prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        if (prevAllocPoint != _allocPoint) {
            updateStakingPool();
        }
    }
```

Listing 3.9: `LPFarm::updateMultiplier()/add()/set()`

```
    function setRewardsPerSecond(uint _rewardsPerSecond) external onlyOwner {
        _updateAllPools();
        uint oldrewardsPerSecond = rewardsPerSecond;
        rewardsPerSecond = _rewardsPerSecond;
        emit NewRewardsPerSecond(rewardsPerSecond, oldrewardsPerSecond);
    }
```

Listing 3.10: `LPFarm::setRewardsPerSecond()`

Secondly, the privileged functions in the `Stake` contract allows for the the `owner` to update the reward multiplier and set the reward amount per second.

```
    function updateMultiplier(uint multiplierNumber) public onlyOwner {
        require(multiplierNumber <= BONUS_MULTIPLIER_MAX, "too large");
        updatePool();
        emit NewMultiplier(multiplierNumber, bounsMultiplier);
        bounsMultiplier = multiplierNumber;
    }
    function setRewardsPerSecond(uint _rewardsPerSecond) external onlyOwner {
        require(_rewardsPerSecond <= REWARDS_PER_MAX, "too large");
        updatePool();
        emit NewRewardsPerSecond(_rewardsPerSecond, rewardsPerSecond);
        rewardsPerSecond = _rewardsPerSecond;

    }
```

Listing 3.11: `Stake::updateMultiplier()/setRewardsPerSecond()`

Lastly, the privileged function `systemStop()` in the `FTPGuardian` contract allows for the `guardian` to pause the lending services. It should be noted that the `FTPGuardian` can be updated by the `owner`.

```
    function setGuardian(address newGuardian) external{
    require(msg.sender == owner,"only owner can call this function");
    require(guardian != newGuardian,"newGuardian can not be same as oldGuardian");
    address oldGuardian = guardian;
    guardian = newGuardian;
    emit NewGuardian(oldGuardian, newGuardian);
  }
    function systemStop(address _unitroller,address _stableunitroller) external {
```

```
    require(msg.sender == guardian,"permission deny");
    IComptroller unitroller = IComptroller(_unitroller);
    IComptroller stableUintroller = IComptroller(_stableunitroller);

    address[] memory ctokens1 = unitroller.getAllMarkets();
    for(uint i = 0; i < ctokens1.length; i++){
      if(!unitroller.mintGuardianPaused(address(ctokens1[i]))){
        unitroller._setMintPaused(ctokens1[i],true);
      }
      if(!unitroller.borrowGuardianPaused(address(ctokens1[i]))){
        unitroller._setBorrowPaused(ctokens1[i],true);
      }
    }
    address[] memory ctokens2 = stableUintroller.getAllMarkets();
    for(uint i = 0; i < ctokens2.length; i++){
      if(!stableUintroller.mintGuardianPaused(address(ctokens2[i]))){
        stableUintroller._setMintPaused(ctokens2[i],true);
      }
      if(!stableUintroller.borrowGuardianPaused(address(ctokens2[i]))){
        stableUintroller._setBorrowPaused(ctokens2[i],true);
      }

    }
    if(!unitroller.transferGuardianPaused()){
      unitroller._setTransferPaused(true);
    }
    if(!unitroller.seizeGuardianPaused()){
      unitroller._setSeizePaused(true);
    }
    if(!stableUintroller.transferGuardianPaused()){
      stableUintroller._setTransferPaused(true);
    }
    if(!stableUintroller.seizeGuardianPaused()){
      stableUintroller._setSeizePaused(true);
    }
  }
```

Listing 3.12: `FTPGuardian::setGuardian()/systemStop()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `owner/guardian` explicit to `Fountain Protocol` users.

Status This issue has been confirmed. The `Fountain` team confirms that the `Timelock` will be set as the owner instead of EOA.

3.8 Proper dsrPerBlock() Calculation

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: DAIInterestRateModelV3
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier, the Fountain protocol is heavily forked from Compound by capitalizing the pooled funds for additional interest. Within the audited codebase, there is a contract DAIInterestRateModelV3, 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 1 second per block on Oasis.

```
    /**
     * @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
            .mul(15); // 15 seconds per block
    }
```

Listing 3.13: DAIInterestRateModelV3::dsrPerBlock()

Note another routine poke() within the same contract shares the same issue.

```
    function poke() public {
        (uint duty, ) = jug.ilks("ETH-A");
        uint stabilityFeePerBlock = duty.add(jug.base()).sub(1e27).mul(1e18).div(1e27).
            mul(15);

        // We ensure the minimum borrow rate >= DSR / (1 - reserve factor)
        baseRatePerBlock = dsrPerBlock().mul(1e18).div(
            assumedOneMinusReserveFactorMantissa);

        // The roof borrow rate is max(base rate, stability fee) + gap, from which we
            derive the slope
        if (baseRatePerBlock < stabilityFeePerBlock) {
```

```
            multiplierPerBlock = stabilityFeePerBlock.sub(baseRatePerBlock).add(
                gapPerBlock).mul(1e18).div(kink);
        } else {
            multiplierPerBlock = gapPerBlock.mul(1e18).div(kink);
        }

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock,
            jumpMultiplierPerBlock, kink);
    }
```

Listing 3.14: `DAIInterestRateModelV3::poke()`

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

Status The issue has been fixed by this commit: `f7a3fc9`.

4 Conclusion

In this audit, we have analyzed the `Fountain` protocol design and implementation. The protocol is designed to be an algorithmic money market that is inspired from `Compound` with the planned deployment on `Oasis`. 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

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

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

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

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

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

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