



SMART CONTRACT AUDIT REPORT

for

ChocolateCash Prototcol



Prepared By: Xiaomi Huang

PeckShield
August 15, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	August 15, 2022	Xuxian Jiang	Final Release
1.0-rc1	August 11, 2022	Xuxian Jiang	Release Candidate #1

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	4
1.1	About ChocolateCash	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	9
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Improved Reentrancy Protection in ContractGuard::onlyOneBlock()	12
3.2	Potential Overflow Mitigation in notifyRewardAmount()	13
3.3	Redundant Logic Removal in LLCUSDTLPTokenSharePool	15
3.4	Simplified Logic in getReward()	16
3.5	Improved Logic in Router::swapAndAddLiquidity()	17
3.6	Trust Issue of Admin Keys	20
4	Conclusion	22
	References	23

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `ChocolateCash` 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 ChocolateCash

`ChocolateCash` is a stable, decentralized cryptocurrency that is designed to expand and contract supply similarly to the way central banks buy and sell fiscal debt to stabilize purchasing power. It remains stable by incentivizing traders to buy and sell in response to changes in demand. These incentives are set up through regular, on-chain auctions of `bond` and `share` tokens, which serve to adjust the stablecoin supply. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ChocolateCash

Item	Description
Name	ChocolateCash
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 15, 2022

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

- <https://github.com/ChocolateCash/ChocolateCash-core.git> (8ed07ca)

And here is the commit ID (after all fixes for the issues found in the audit have been checked in) as well as the related deployed token contracts.

- <https://github.com/ChocolateCash/ChocolateCash-core.git> (bc95650)
- CHC: <https://www.hecoinfo.com/address/0x7A5E553888AfB3a44C3449fe2C8e1592957c678C>
- CHS: <https://www.hecoinfo.com/address/0x7F4D7F2f19E993E713FE85e877f34C949178Bf55>
- CHB: <https://www.hecoinfo.com/address/0xf6850A259FB66484EAd171ADc3dAf86b1284dA53>

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

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

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

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




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.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `ChocolateCash` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	3	
Undetermined	1	
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 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Audit Findings of ChocolateCash Protocol

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Improved Reentrancy Protection in <code>ContractGuard::onlyOneBlock()</code>	Time and State	Confirmed
PVE-002	Medium	Potential Overflow Mitigation in <code>notifyRewardAmount()</code>	Numeric Errors	Confirmed
PVE-003	Low	Redundant Logic Removal in <code>LLCUS-DTLPTokenSharePool</code>	Coding Practices	Confirmed
PVE-004	Low	Simplified Logic in <code>getReward()</code>	Numeric Errors	Confirmed
PVE-005	Low	Improved Logic in <code>Router::swapAndAddLiquidity()</code>	Coding Practices	Confirmed
PVE-006	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 Improved Reentrancy Protection in `ContractGuard::onlyOneBlock()`

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `ContractGuard`
- Category: Time and State [8]
- 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` [14] exploit, as well as the `Uniswap/Lendf.Me` hack [13].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `ContractGuard` as an example, the `onlyOneBlock()` modifier (see the code snippet below) is provided to ensure the caller can interact with the contract at most once within a block. However, the invocation of an external contract in the middle requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 24) start before effecting the update on internal states (lines 26 – 27), 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 various functions even protected with this modifier.

```
14     modifier onlyOneBlock() {  
15         require(  
16             !checkSameOriginReentranted(),
```

```

17         'ContractGuard: one block, one function'
18     );
19     require(
20         !checkSameSenderReentranted(),
21         'ContractGuard: one block, one function'
22     );
23
24     _;
25
26     _status[block.number][tx.origin] = true;
27     _status[block.number][msg.sender] = true;
28 }

```

Listing 3.1: ContractGuard::onlyOneBlock()

Recommendation Apply proper reentrancy prevention by following the checks-effects-interactions principle in the above `onlyOneBlock` modifier.

Status The issue has been confirmed.

3.2 Potential Overflow Mitigation in `notifyRewardAmount()`

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

The `ChocolateCash` protocol has a built-in incentivizer mechanism, which is based on the popular `StakingRewards` from `Synthetix`. In this section, we focus on a routine, i.e., `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token and it is part of the `updateReward` modifier that would be invoked up-front for almost every public function in `Boardroom` to update and use the latest reward rate.

The reason is due to the known potential overflow pitfall when a new oversized reward amount is added into the pool. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines 61 – 67), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardDistribution` (through the `notifyRewardAmount()` function).

```

52     function lastTimeRewardApplicable() public view returns (uint256) {
53         return Math.min(block.timestamp, periodFinish);
54     }

```

```

55
56     function rewardPerToken() public view returns (uint256) {
57         if (totalSupply() == 0) {
58             return rewardPerTokenStored;
59         }
60         return
61             rewardPerTokenStored.add(
62                 lastTimeRewardApplicable()
63                     .sub(lastUpdateTime)
64                     .mul(rewardRate)
65                     .mul(1e18)
66                     .div(totalSupply()));
67     };
68 }

```

Listing 3.2: Boardroom::rewardPerToken()

```

105     function notifyRewardAmount(uint256 reward)
106         override
107         external
108         onlyRewardDistribution
109         updateReward(address(0))
110     {
111         if (block.timestamp >= periodFinish) {
112             rewardRate = reward.div(DURATION);
113         } else {
114             uint256 remaining = periodFinish.sub(block.timestamp);
115             uint256 leftover = remaining.mul(rewardRate);
116             rewardRate = reward.add(leftover).div(DURATION);
117         }
118         lastUpdateTime = block.timestamp;
119         periodFinish = block.timestamp.add(DURATION);
120         emit RewardAdded(reward);
121     }

```

Listing 3.3: Boardroom::notifyRewardAmount()

Apparently, this issue is made possible if the reward amount is given as the argument to `notifyRewardAmount()` such that the calculation of `rewardRate.mul(1e18)` always overflows, hence locking all deposited funds! Note that an authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `rewardDistribution` address is able to call `notifyRewardAmount()` and this address is set by the owner. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Be consistent and sufficient in mitigating the potential overflow risk in all affected pools, including `Boardroom`, `MigrationPool`, `ReferPool`, and `LLCUSDTokenSharePool`.

Status This issue has been confirmed as the team clarifies no large reward will be used to cause

overflow.

3.3 Redundant Logic Removal in LLCUSDTLPTokenSharePool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

The ChocolateCash 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 LLCUSDTLPTokenSharePool 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 LLCUSDTLPTokenSharePool::stake() function, there is a redundant check on the referer presence for the calling user. Specifically, the following stake() function requires the calling user has her referer. And it also has an if-statement that checks the refer presence (line 106). Apparently, the check on the if-statement always yields true, which suggests the simplification on removing the if-condition.

```

95 // stake visibility is public as overriding LPTokenWrapper's stake() function
96 function stake(Kind kind, uint amount)
97     public
98     updateReward(msg.sender)
99     checkhalve
100    checkStart
101    {
102        require(amount > 0, 'Cannot stake 0');
103        require(refer.hasReferrer(msg.sender), "can not stake without referer");
104        super._stake(kind, amount);
105        address _referer;
106        if(refer.hasReferrer(msg.sender)){
107            _referer = refer.referrer(msg.sender);
108            IPool(referPool).stake(_referer, amount);
109            if(refer.hasReferrer(_referer)){
110                _referer = refer.referrer(_referer);
111                IPool(referPool).stake(_referer, amount);
112            }
113        }
114
115        lpt.safeTransferFrom(msg.sender, address(this), amount);

```

116

}

Listing 3.4: `LLCUSDTLPTokenSharePool::stake()`

Moreover, the `Treasury` contract defines a few unused events, which can be removed as well. Examples include `ContributionPoolChanged`, `ContributionPoolRateChanged`, and `ContributionPoolFunded`.

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

Status This issue has been confirmed.

3.4 Simplified Logic in `getReward()`

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

As mentioned earlier, the `ChocolateCash` protocol has a built-in incentivizer mechanism that is based on the popular `StakingRewards` from `Synthetix`. In particular, there is a related `getReward()` routine, which is intended to obtain the calling user's staking rewards. Specifically, the current logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 63).

```

141     function getReward() public updateReward(msg.sender) checkhalve checkStart {
142         uint256 reward = earned(msg.sender);
143         if (reward > 0) {
144             rewards[msg.sender] = 0;
145             share.safeTransfer(msg.sender, reward);
146             emit RewardPaid(msg.sender, reward);
147         }
148     }

```

Listing 3.5: `LLCUSDTLPTokenSharePool::getReward()`


```

59     modifier updateReward(address account) {
60         rewardPerTokenStored = rewardPerToken();
61         lastUpdateTime = lastTimeRewardApplicable();
62         if (account != address(0)) {
63             rewards[account] = earned(account);
64             userRewardPerTokenPaid[account] = rewardPerTokenStored;
65         }
66         -;
67     }

```

Listing 3.6: LLCUSDTLPTokenSharePool::updateReward()

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the caller `msg.sender`. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the reward variable (line 63).

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```

141     function getReward() public updateReward(msg.sender) checkhalve checkStart {
142         uint256 reward = rewards[msg.sender];
143         if (reward > 0) {
144             rewards[msg.sender] = 0;
145             share.safeTransfer(msg.sender, reward);
146             emit RewardPaid(msg.sender, reward);
147         }
148     }

```

Listing 3.7: LLCUSDTLPTokenSharePool::getReward()

Status This issue has been confirmed.

3.5 Improved Logic in Router::swapAndAddLiquidity()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Router
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

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 analyze 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>).

```

194  /**
195  * @dev Approve the passed address to spend the specified amount of tokens on behalf
      of msg.sender.
196  * @param _spender The address which will spend the funds.
197  * @param _value The amount of tokens to be spent.
198  */
199  function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201      // To change the approve amount you first have to reduce the addresses'
202      // allowance to zero by calling 'approve(_spender, 0)' if it is not
203      // already 0 to mitigate the race condition described here:
204      // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205      require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

207      allowed[msg.sender][_spender] = _value;
208      Approval(msg.sender, _spender, _value);
209  }

```

Listing 3.8: USDT Token Contract

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

```

38  /**
39  * @dev Deprecated. This function has issues similar to the ones found in
40  * {IERC20-approve}, and its usage is discouraged.
41  *
42  * Whenever possible, use {safeIncreaseAllowance} and
43  * {safeDecreaseAllowance} instead.
44  */
45  function safeApprove(
46      IERC20 token,
47      address spender,
48      uint256 value
49  ) internal {
50      // safeApprove should only be called when setting an initial allowance,
51      // or when resetting it to zero. To increase and decrease it, use
52      // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
53      require(
54          (value == 0) || (token.allowance(address(this), spender) == 0),

```

```

55         "SafeERC20: approve from non-zero to non-zero allowance"
56     );
57     _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
58         spender, value));

```

Listing 3.9: SafeERC20::safeApprove()

In current implementation, if we examine the Router::swapAndAddLiquidity() routine that is designed to swap and add the intended liquidity. To accommodate the specific idiosyncrasy, there is a need to use safeApprove(), instead of approve() (line 31).

```

21     function swapAndAddLiquidity(uint amount) public {
22         require(amount > 0, "amount must > 0");
23
24         usdt.safeTransferFrom(msg.sender, address(this), amount);
25
26         //
27         uint usdtAmount = amount.div(2);
28         address[] memory path = new address[](2);
29         path[0] = address(usdt);
30         path[1] = address(token);
31         usdt.approve(address(router), usdtAmount);
32         uint[] memory tokenAmounts = router.swapExactTokensForTokens(usdtAmount, 0,
33             path, address(this), block.timestamp);
34         uint targetAmount = tokenAmounts[1];
35
36         usdt.approve(address(router), usdtAmount);
37         token.approve(address(router), targetAmount);
38         (uint amountInUsdt, uint amountInToken,) = router.addLiquidity(address(usdt),
39             address(token), usdtAmount, targetAmount, 0, 0, msg.sender, block.timestamp);
40
41         //
42         if(usdtAmount > amountInUsdt) {
43             usdt.safeTransfer(msg.sender, usdtAmount.sub(amountInUsdt));
44         }
45
46         if(targetAmount > amountInToken) {
47             token.transfer(msg.sender, targetAmount.sub(amountInToken));
48         }
49     }

```

Listing 3.10: Router::swapAndAddLiquidity()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve().

Status This issue has been confirmed.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `ChocolateCash` contract, there is a privileged `admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., execute the timelocked tasks, etc). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

To elaborate, we show below example privileged routines in multiple contracts. These routines allow the `admin` account to queue and execute protocol-sensitive tasks and make adjustments to the protocol dynamics.

```

175     function executeTransaction(
176         address target,
177         uint256 value,
178         string memory signature,
179         bytes memory data,
180         uint256 eta
181     ) public payable returns (bytes memory) {
182         require(
183             msg.sender == admin,
184             'Timelock::executeTransaction: Call must come from admin.'
185         );
186
187         bytes32 txHash = keccak256(
188             abi.encode(target, value, signature, data, eta)
189         );
190         require(
191             queuedTransactions[txHash],
192             "Timelock::executeTransaction: Transaction hasn't been queued."
193         );
194         ...
195     }

```

Listing 3.11: `Timelock::executeTransaction()`

It would be worrisome if the privileged `admin` account is a plain EOA account. Note that 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.

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

Status This issue has been confirmed and the team has decided not to use EOA to be the privileged account.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `ChocolateCash` protocol, which is a stable, decentralized cryptocurrency designed to expand and contract supply similarly to the way central banks buy and sell fiscal debt to stabilize purchasing power. It remains stable by incentivizing traders to buy and sell in response to changes in demand. These incentives are set up through regular, on-chain auctions of `bond` and `share` tokens, which serve to adjust the stablecoin supply. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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



References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-190: Integer Overflow or Wraparound. <https://cwe.mitre.org/data/definitions/190.html>.
- [3] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [4] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.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 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: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.
- [13] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [14] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

