SMART CONTRACT AUDIT REPORT

for

DarkCrypto

Prepared By: Yiqun Chen

PeckShield

February 26, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	February 26, 2022	Jing Wang	Final Release
1.0-rc	February 24, 2022	Jing Wang	Release Candidate

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 DarkCrypto 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 DarkCrypto

DarkCrypto protocol is an ecosystem running around DARK (an algorithmic token pegged to CRO on Cronos chain). The DARK token involves a solution that can adjust the stablecoin's supply deterministically to move the price of the stablecoin in the direction of a target price, which brings the desired programmability and interoperability to DeFi. The basic information of the DarkCrypto protocol is as follows:

Table 1.1: Basic Information of The DarkCrypto Protocol

Item	Description
Issuer	DarkCrypto
Website	https://www.darkcrypto.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 26, 2022

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

- https://github.com/darkcryptofinance/darkcrypto-contracts.git (fee5be8)

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

- https://github.com/darkcryptofinance/darkcrypto-contracts.git (50197b1)

1.2 About PeckShield

PeckShield Inc. [14] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [13]:

- 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) [12], 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 `DarkCrypto` 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	1	
Medium	4	
Low	4	
Informational	0	
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 4 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key DarkCrypto Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Tax-Evasion By Calling Transfer() Directly	Business Logic	Fixed
PVE-002	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-003	Medium	Potential Underflow For tierId	Security Features	Fixed
PVE-004	Low	Incompatibility with Deflationary Tokens	Time and State	Fixed
PVE-005	Low	Potential Reentrancy in withdraw()	Time and State	Fixed
PVE-006	Low	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Partially Fixed
PVE-007	Medium	Proper Refund of The Excess ETH	Time and State	Fixed
PVE-008	Medium	Inconsistent Dark Amount Calculation in getBurnableDarkLeft() and redeemBonds()	Business Logic	Fixed
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Tax-Evasion By Calling Transfer() Directly

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

Description

Within the `DarkCrypto` protocol, `Dark` is an algorithmic token pegged to `CRO` and is designed to be used as a medium of exchange. The built-in stability mechanism in the protocol deterministically expands and contracts the `DARK` supply to maintain the `DARK`'s peg to 1 `CRO` in the long run. At the same time, `DARK` is also used in the tax system where a tax is charged on selling `DARKs`.

In the following, we analyze how the tax is applied in the implementation. Specifically, we show below the code snippet from the `DarkCrypto::transferFrom()` routine. This routine is called when the `spender` transfers `DARK` from the `sender` to the `recipient`. Inside this routine, a helper routine `_transferWithTax()` (line 208) is used to collect the tax.

```
189     function transferFrom(
190         address sender,
191         address recipient,
192         uint256 amount
193     ) public override returns (bool) {
194         uint256 currentTaxRate = 0;
195         bool burnTax = false;

197         if (autoCalculateTax) {
198             uint256 currentDarkPrice = _getDarkPrice();
199             currentTaxRate = _updateTaxRate(currentDarkPrice);
200             if (currentDarkPrice < burnThreshold) {
201                 burnTax = true;
202             }
203         }
```

```
        if (currentTaxRate == 0  excludedAddresses[sender]) {
            _transfer(sender, recipient, amount);
        } else {
            _transferWithTax(sender, recipient, amount, burnTax);
        }

        _approve(sender, _msgSender(), allowance(sender, _msgSender()).sub(amount, "ERC20: transfer amount exceeds allowance"));
        return true;
    }
```

Listing 3.1: DarkCrypto::transferFrom()

However, we notice an interesting tax-evasion issue that may prevent the tax-collection from properly functioning. Specifically, the tax-evasion issue comes from the fact that the tax-collection helper routine `_transferWithTax()` is only applied in the `transferFrom()` routine. If the `sender` calls the `transfer()` routine to directly transfer `DARK` to the `spender` and then let the `spender` to do another `transfer()` to transfer `DARK` to the `recipient`, the current tax-collection enforcement is bypassed.

```
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }
```

Listing 3.2: DarkCrypto::transfer()

Recommendation Apply the tax-collection helper routine `_transferWithTax()` in the `transfer()` routine.

Status This issue has been confirmed by the team. The team clarifies they will not apply tax for the `DARK` token. Also, the team set the `TaxOffice` to a dead address by the following transaction `0xd7c3...80cb`. The only way to switch `TaxOffice` back would be let the owner `TimeLock` to set a new `Operator` and then set the new `TaxOffice`.

3.2 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: SkyRewardPool
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The DarkCrypto protocol provides incentive mechanisms that reward the staking of supported assets. 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 added routine add(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
    function add(
        uint256 _allocPoint,
        IERC20 _token,
        bool _withUpdate,
        uint256 _lastRewardTime
    ) public onlyOperator {
        checkPoolDuplicate(_token);
        if (_withUpdate) {
            massUpdatePools();
        }
        ...
    }
```

Listing 3.3: SkyRewardPool::add()

If the call to massUpdatePools() is not immediately invoked before adding 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 onlyOperator modifier), which greatly alleviates the concern.

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

```
    function add(
        uint256 _allocPoint,
```

```
        IERC20 _token,
        bool _withUpdate,
        uint256 _lastRewardTime
    ) public onlyOperator {
        checkPoolDuplicate(_token);
        massUpdatePools();
        ...
    }
```

Listing 3.4: `SkyRewardPool::add()`

Status The issue has been fixed by this commit: `50197b1`.

3.3 Potential Underflow For tierId

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `DarkCrypto`
- Category: Security Features [7]
- CWE subcategory: CWE-282 [2]

Description

As mentioned in Section 3.1, a tax is charged on selling `DARK` token. To support different tax tires, the token contract defines two arrays (`taxTiersTwaps[]` and `taxTiersRates[]`) and a helper routine (`_updateTaxRate()`) to update tax rate automatically.

To illustrate, we show below the `_updateTaxRate()` helper routine. This helper routine is internally used to update tax by the price of `DARK`. Specifically, it is called in every single `transferFrom()` operation.

```
    function _updateTaxRate(uint256 _darkPrice) internal returns (uint256){
        if (autoCalculateTax) {
            for (uint8 tierId = uint8(getTaxTiersTwapsCount()).sub(1); tierId >= 0; --tierId) {
                if (_darkPrice >= taxTiersTwaps[tierId]) {
                    require(taxTiersRates[tierId] < 10000, "tax equal or bigger to 100%");
                    taxRate = taxTiersRates[tierId];
                    return taxTiersRates[tierId];
                }
            }
        }
    }
```

Listing 3.5: DarkCrypto::_updateTaxRate()

The analysis with the above helper routine shows there is a loop from the end to the start of the taxTiersTwaps[] array to find who is the first slot that has a value smaller than _darkPrice. The above algorithm works on the assumption that at least taxTiersTwaps[0] will be smaller than or equal to _darkPrice so the loop could stop when tierId equals to 0. However, there is no guarantee that the operation of --tieId will not cause underflow and lead to a infinite loop, which finally cause a revert transaction in every single transferFrom() operation.

Recommendation Apply the SafeMath to block unintended underflow.

Status This issue has been confirmed by the team. The team clarifies they will not apply tax for the DARK token. Also, the team set the TaxOffice to a dead address by the following transaction 0xd7c3...80cb. The only way to switch TaxOffice back would be let the owner TimeLock to set a new Operator and then set the new TaxOffice.

3.4 Incompatibility with Deflationary Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SkyRewardPool
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the DarkCrypto protocol, the SkyRewardPool contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., deposit(), accepts asset transfer-in and records the depositor's balance. Another interface, i.e, withdraw(), allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., deposit() and withdraw(), the contract using the safeTransferFrom() routine to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    // Deposit LP tokens.
    function deposit(uint256 _pid, uint256 _amount) public {
        address _sender = msg.sender;
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 _pending = user.amount.mul(pool.accSkyPerShare).div(1e18).sub(user.
                rewardDebt);
```

```
            if (_pending > 0) {
                safeSkyTransfer(_sender, _pending);
                emit RewardPaid(_sender, _pending);
            }
        }
        if (_amount > 0) {
            pool.token.safeTransferFrom(_sender, address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accSkyPerShare).div(1e18);
        emit Deposit(_sender, _pid, _amount);
    }

    // Withdraw LP tokens.
    function withdraw(uint256 _pid, uint256 _amount) public {
        address _sender = msg.sender;
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 _pending = user.amount.mul(pool.accSkyPerShare).div(1e18).sub(user.
            rewardDebt);
        if (_pending > 0) {
            safeSkyTransfer(_sender, _pending);
            emit RewardPaid(_sender, _pending);
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.token.safeTransfer(_sender, _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accSkyPerShare).div(1e18);
        emit Withdraw(_sender, _pid, _amount);
    }
```

Listing 3.6: `SkyRewardPool::deposit()`and `SkyRewardPool::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accSkyPerShare` via dividing `_skyReward` by `tokenSupply`, where the `tokenSupply` is derived from `balanceOf(address(this))` (line 177). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may give a big `pool.accSkyPerShare` as the final result, which dramatically inflates the pool's reward.

```
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.timestamp <= pool.lastRewardTime) {
            return;
        }
        uint256 tokenSupply = pool.token.balanceOf(address(this));
        if (tokenSupply == 0) {
            pool.lastRewardTime = block.timestamp;
            return;
        }
        if (!pool.isStarted) {
            pool.isStarted = true;
            totalAllocPoint = totalAllocPoint.add(pool.allocPoint);
        }
        if (totalAllocPoint > 0) {
            uint256 _generatedReward = getGeneratedReward(pool.lastRewardTime, block.timestamp);
            uint256 _skyReward = _generatedReward.mul(pool.allocPoint).div(totalAllocPoint);
            pool.accSkyPerShare = pool.accSkyPerShare.add(_skyReward.mul(1e18).div(tokenSupply));
        }
        pool.lastRewardTime = block.timestamp;
    }
```

Listing 3.7: `SkyRewardPool::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `DarkCrypto` protocol for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status The issue has been fixed by this commit: `50197b1`.

3.5 Potential Reentrancy in withdraw()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SkyRewardPool
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

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

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `SkyRewardPool` as an example, the `withdraw()` function (see the code snippet below) is provided to withdraw `LP` tokens from the pool. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (lines 224 and 229) starts before effecting the update on internal states (line 231), 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.

```
216     function withdraw(uint256 _pid, uint256 _amount) public {
217         address _sender = msg.sender;
218         PoolInfo storage pool = poolInfo[_pid];
219         UserInfo storage user = userInfo[_pid][_sender];
220         require(user.amount >= _amount, "withdraw: not good");
221         updatePool(_pid);
222         uint256 _pending = user.amount.mul(pool.accSkyPerShare).div(1e18).sub(user.
                rewardDebt);
223         if (_pending > 0) {
224             safeSkyTransfer(_sender, _pending);
225             emit RewardPaid(_sender, _pending);
226         }
227         if (_amount > 0) {
228             user.amount = user.amount.sub(_amount);
229             pool.token.safeTransfer(_sender, _amount);
230         }
231         user.rewardDebt = user.amount.mul(pool.accSkyPerShare).div(1e18);
232         emit Withdraw(_sender, _pid, _amount);
```

```
    }
```

Listing 3.8: `SkyRewardPool::withdraw()`

Note another routine `deposit()` shares the same issue.

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

Status The issue has been fixed by this commit: `50197b1`.

3.6 Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- 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 `transfer()` 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.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
```

```
    }
```

Listing 3.9: USDT Token Contract

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address recipient, uint256 amount)external returns (bool)`. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `governanceRecoverUnsupported()` routine in the `DarkCryptoShare` contract. If the `USDT` token is given as the routine's argument, i.e., `_token`, the unsafe version of `_token.transfer(_to, _amount)` (line 113) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
108     function governanceRecoverUnsupported(
109         IERC20 _token,
110         uint256 _amount,
111         address _to
112     ) external onlyOperator {
113         _token.transfer(_to, _amount);
114     }
```

Listing 3.10: DarkCryptoShare::governanceRecoverUnsupported()

Note that the same issue exists in the `addLiquidityTaxFree()` routine from the `TaxOffice` contract. Also, the `_approveTokenIfNeeded()` helper from the `TaxOffice` contract shares the same issue, which may revert a number of calling routines.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been partially fixed by this commit: `50197b1`.

3.7 Proper Refund of The Excess ETH

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TaxOffice`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

In `DarkCrypto`, there is a contract `TaxOffice` which provides a number of convenience routines for liquidation addition, e.g., `addLiquidityTaxFree()` and `addLiquidityETHTaxFree()`. During the analysis of these convenience routines, we notice that `addLiquidityETHTaxFree()` does not refund the excess `ETH` properly.

To elaborate, we show below the implementation of `addLiquidityTaxFree()`. This routine receives `ETH` and tokens from the caller, provides them to the `uniRouter` to add liquidity. The `uniRouter` should refund the excess `ETH` to the `TaxOffice` contract.

```
    function addLiquidityETHTaxFree(
        uint256 amtDark,
        uint256 amtDarkMin,
        uint256 amtFtmMin
    )
        external
        payable
        returns (
            uint256,
            uint256,
            uint256
        )
    {
        require(amtDark != 0 && msg.value != 0, "amounts can't be 0");
        _excludeAddressFromTax(msg.sender);

        IERC20(dark).transferFrom(msg.sender, address(this), amtDark);
        _approveTokenIfNeeded(dark, uniRouter);

        _includeAddressInTax(msg.sender);

        uint256 resultAmtDark;
        uint256 resultAmtFtm;
        uint256 liquidity;
        (resultAmtDark, resultAmtFtm, liquidity) = IUniswapV2Router(uniRouter).
            addLiquidityETH{value: msg.value}(
            dark,
            amtDark,
            amtDarkMin,
```

```
            amtFtmMin,
            msg.sender,
            block.timestamp
        );

        if(amtDark.sub(resultAmtDark) > 0) {
            IERC20(dark).transfer(msg.sender, amtDark.sub(resultAmtDark));
        }
        return (resultAmtDark, resultAmtFtm, liquidity);
    }
```

Listing 3.11: `TaxOffice::addLiquidityETHTaxFree()`

However, it comes to our attention that in the current implementation, the excess `ETH` returned from the router to the `TaxOffice` contract is not sent back to the caller. This will cause `msg.value - resultAmtFtm` amount of `ETH` left in the contract.

Recommendation Revise the above routine to properly return the excess ETH back to the user.

Status This issue has been confirmed by the team. The team clarifies they will not apply tax for the `DARK` token. Also, the team set the `TaxOffice` to a dead address by the following transaction `0xd7c3...80cb`. The only way to switch `TaxOffice` back would be let the owner `TimeLock` to set a new `Operator` and then set the new `TaxOffice`.

3.8 Inconsistent Dark Amount Calculation in getBurnableDarkLeft() and redeemBonds()

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

Description

In the `Treasury` contract, the `getBurnableDarkLeft()` viewer function allows the caller to get the `_burnableDarkLeft`. As shown in the following code snippets, the `_burnableDarkLeft` takes the minimum value between `epochSupplyContractionLeft` and `_maxBurnableDark`, where `_maxBurnableDark` is derived from `_maxMintableBond.mul(_darkPrice).div(1e18)` (line 189).

```
181     function getBurnableDarkLeft() public view returns (uint256 _burnableDarkLeft) {
182         uint256 _darkPrice = getDarkPrice();
183         if (_darkPrice <= darkPriceOne) {
```

```
            uint256 _darkSupply = getDarkCirculatingSupply();
            uint256 _bondMaxSupply = _darkSupply.mul(maxDebtRatioPercent).div(10000);
            uint256 _bondSupply = IERC20(light).totalSupply();
            if (_bondMaxSupply > _bondSupply) {
                uint256 _maxMintableBond = _bondMaxSupply.sub(_bondSupply);
                uint256 _maxBurnableDark = _maxMintableBond.mul(_darkPrice).div(1e18);
                _burnableDarkLeft = Math.min(epochSupplyContractionLeft,
                    _maxBurnableDark);
            }
        }
    }
```

Listing 3.12: `Treasury::getBurnableDarkLeft()`

On the other hand, the `redeemBonds()` function in the same contract also calculates the amount of `DARK` could be redeemed. As shown in the following code snippets, the `_darkAmount` is derived from `_bondAmount.mul(_rate).div(1e18)` (line 466). The inconsistent `Dark` amount calculations between these two functions may introduce unexpected result.

```
453     function redeemBonds(uint256 _bondAmount, uint256 targetPrice) external onlyOneBlock
             checkCondition checkOperator {
454         require(_bondAmount > 0, "Treasury: cannot redeem bonds with zero amount");

456         uint256 darkPrice = getDarkPrice();
457         require(darkPrice == targetPrice, "Treasury: DARK price moved");
458         require(
459             darkPrice > darkPriceCeiling, // price > $1.01
460             "Treasury: darkPrice not eligible for bond purchase"
461         );

463         uint256 _rate = getBondPremiumRate();
464         require(_rate > 0, "Treasury: invalid bond rate");

466         uint256 _darkAmount = _bondAmount.mul(_rate).div(1e18);
467         require(IERC20(dark).balanceOf(address(this)) >= _darkAmount, "Treasury:
                treasury has no more budget");

469         seigniorageSaved = seigniorageSaved.sub(Math.min(seigniorageSaved, _darkAmount))
                ;

471         IBasisAsset(light).burnFrom(msg.sender, _bondAmount);
472         IERC20(dark).safeTransfer(msg.sender, _darkAmount);

474         _updateDarkPrice();

476         emit RedeemedBonds(msg.sender, _darkAmount, _bondAmount);
477     }
```

Listing 3.13: `Treasury::redeemBonds()`

Recommendation Fix the `DARK` amount calculation in the `getBurnableDarkLeft()` routine.

Status The issue has been fixed by this commit: `50197b1`.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

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

To elaborate, we show below several privileged functions in the `Treasury` contract, which allows the `Operator` to configure some critical system-wide parameters or settings.

```
    function setBoardroom(address _boardroom) external onlyOperator {
        boardroom = _boardroom;
    }

    function setDarkOracle(address _darkOracle) external onlyOperator {
        darkOracle = _darkOracle;
    }

    function setDarkPriceCeiling(uint256 _darkPriceCeiling) external onlyOperator {
        require(_darkPriceCeiling >= darkPriceOne && _darkPriceCeiling <= darkPriceOne.
            mul(120).div(100), "out of range"); // [$1.0, $1.2]
        darkPriceCeiling = _darkPriceCeiling;
    }

    function boardroomSetOperator(address _operator) external onlyOperator {
        IBoardroom(boardroom).setOperator(_operator);
    }
```

Listing 3.14: `Treasury.sol`

It is 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.

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 The issue has been confirmed and the team moves the Operator account to a Timelock contract. Moreover, the team clarifies they will implement the DAO governance when the protocol becomes mature and stable.

4 | Conclusion

In this audit, we have analyzed the DarkCrypto protocol design and implementation. DarkCrypto protocol is an ecosystem running around DARK (an algorithmic token pegged to CRO on Cronos chain). During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

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

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

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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