SMART CONTRACT AUDIT REPORT

for

DarkNess Dollar

Prepared By: Patrick Lou

PeckShield

March 31, 2022

Document Properties

Client	DarkNess Finance
Title	Smart Contract Audit Report
Target	DarkNess Dollar
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 31, 2022	Xiaotao Wu	Final Release
1.0-rc	March 30, 2022	Xiaotao Wu	Release Candidate

Contact

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

Name	Patrick Lou
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 DarkNess Dollar 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About DarkNess Dollar

DarkNess Dollar is a stablecoin with parts of its supply backed by collateral and parts of the supply algorithmic. The ratio of collateralized and algorithmic supply depends on the market's pricing of the DarkNess Dollar stablecoin, i.e., DUSD. If DUSD is trading at above $1, the protocol decreases the collateral ratio. If DUSD is trading at under $1, the protocol increases the collateral ratio. DarkNess Dollar aims to create a highly scalable, trustful, extremely stable, and ideologically pure on-chain money.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of DarkNess Dollar

Item	Description
Name	DarkNess Finance
Website	https://www.darkness.finance
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 31, 2022

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

- https://github.com/darkcryptofi/darknessdollar-contracts.git (3d6bf7d)

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

- https://github.com/darkcryptofi/darknessdollar-contracts.git (e6b3918)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. 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 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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `DarkNess Dollar` smart contracts. 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 logic, 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	1	■
Informational	2	■ ■
Total	5	

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 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, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key DarkNess Dollar Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inconsistent Handling in Pool::trimExtraToTreasury()	Business Logic	Fixed
PVE-002	Low	Potential Sandwich/MEV Attack In CollateralReserve::_sellSharesToUsdc()	Time and State	Confirmed
PVE-003	Informational	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-004	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Inconsistent Handling in Pool::trimExtraToTreasury()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Pool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Pool` contract provides a public `trimExtraToTreasury()` function to transfer extra `collaterals[0]/dark/share` assets from reserve to `profitSharingFund_`. Our analysis with this routine shows the current value assigned to the temporary variable `_mainCollateralBal` is inconsistent with the variable definition.

To elaborate, we show below its code snippet. Specifically, the value assigned to the temporary variable `_mainCollateralBal` should be `_treasury.globalCollateralBalance(0)`, instead of current `_treasury.globalCollateralValue(0).div(10 ** missing_decimals[0])` (line 669).

```
658     function trimExtraToTreasury() public returns (uint256 _collateralAmount, uint256
            _darkAmount, uint256 _shareAmount) {
659         uint256 _collateral_price = getCollateralPrice(0);
660         uint256 _total_dollar_FullValue = IERC20(dollar).totalSupply().mul(
                _collateral_price).div(PRICE_PRECISION);
661         ITreasury _treasury = ITreasury(treasury);
662         uint256 _totalCollateralValue = _treasury.globalCollateralTotalValue();
663         uint256 _dark_bal = _treasury.globalDarkBalance();
664         uint256 _share_bal = _treasury.globalShareBalance();
665         address _profitSharingFund = _treasury.profitSharingFund();
666         if (_totalCollateralValue > _total_dollar_FullValue) {
667             _collateralAmount = _totalCollateralValue.sub(_total_dollar_FullValue).div
                    (10 ** missing_decimals[0]).mul(PRICE_PRECISION).div(_collateral_price);
668             if (_collateralAmount > 0) {
669                 uint256 _mainCollateralBal = _treasury.globalCollateralValue(0).div(10
                        ** missing_decimals[0]);
```

```
670                 if (_collateralAmount > _mainCollateralBal) _collateralAmount = _mainCollateralBal;
671                 _requestTransferFromReserve(collaterals[0], _profitSharingFund, _collateralAmount);
672             }
673             if (_dark_bal > 0) {
674                 _darkAmount = _dark_bal;
675                 _requestTransferFromReserve(dark, _profitSharingFund, _darkAmount);
676             }
677             if (_share_bal > 0) {
678                 _shareAmount = _share_bal;
679                 _requestTransferFromReserve(share, _profitSharingFund, _shareAmount);
680             }
681         } else {
682             ...
683         }
684     }
```

Listing 3.1: `Pool::trimExtraToTreasury()`

Recommendation Assign the correct value to the variable `_mainCollateralBal` (line 669) for above mentioned function.

Status This issue has been fixed in this commit: `e6b3918`.

3.2 Potential Sandwich/MEV Attack In CollateralReserve::_sellSharesToUsdc()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CollateralReserve`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

While examining the `CollateralReserve` contract, we notice there is one function that can be improved with slippage control. To elaborate, we show below the related code snippet of the `CollateralReserve` contract. According to the design, the `_sellSharesToUsdc()` function is used to swap `share` to `USDC`. In the function, the `swapExactTokensForTokens()` function of `UniswapV2` is called (line 113) to swap the exact `share` amount to `USDC`. However, we observe the second input `amountOutMin` parameter is assigned to `1`, which means this transaction does not specify valid restriction on possible slippage and is therefore vulnerable to possible front-running attacks.

```
    function _sellSharesToUsdc(uint256 _amount) internal {
        if (_amount > maxShareAmountToSell) {
            _amount = maxShareAmountToSell;
        }
        uint256 _shareBal = IERC20(share).balanceOf(address(this));
        if (_amount > _shareBal) {
            _amount = _shareBal;
        }
        if (_amount == 0) return;
        IERC20(share).safeIncreaseAllowance(router, _amount);
        uint256 _before = IERC20(usdc).balanceOf(address(this));
        IUniswapV2Router(router).swapExactTokensForTokens(_amount, 1,
            shareToUsdcRouterPath, address(this), block.timestamp.add(1800));
        emit SwapToken(share, usdc, _amount, _after.sub(_before));
    }
```

Listing 3.2: `CollateralReserve::_sellSharesToUsdc()`

Recommendation Improve the above-mentioned function by adding necessary valid slippage control.

Status This issue has been confirmed. The `DarkNess Dollar` team confirms that the extra share is to be sold to get `USDC` for reserve and the swap might get reverted and then hang the whole procedure if an slippage is set.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.3: ZRX.sol

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. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In current implementation, if we examine the `Treasury::rescueStuckErc20()` routine that is designed to withdraw `token` from the contract. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 373).

```
372    function rescueStuckErc20(address _token) external onlyOwner {
373        IERC20(_token).transfer(owner(), IERC20(_token).balanceOf(address(this)));
374    }
```

Listing 3.4: `Treasury::rescueStuckErc20()`

Note that a number of routines can be similarly improved, including `NessToken::rescueStuckErc20()`, `Pool::rescueStuckErc20()`, `CollateralReserve::rescueStuckErc20()`, `Dollar::rescueStuckErc20()`, `TreasuryPolicy::rescueStuckErc20()`, and `OracleAssetToUSDC::rescueStuckErc20()`.

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

Status This issue has been fixed in this commit: `e6b3918`.

3.4 Meaningful Events For Important State Changes

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

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `NessToken` contract as an example. While examining the events that reflect the `NessToken` dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the `setMarketingFund()/setAdvisorFund()` are being called, there are no corresponding events being emitted to reflect the occurrence of `setMarketingFund()/setAdvisorFund()`.

```
    function setMarketingFund(address _marketingFund) external onlyOwner {
        require(_marketingFund != address(0), "zero");
        marketingFund = _marketingFund;
    }

    function setAdvisorFund(address _advisorFund) external onlyOwner {
        require(_advisorFund != address(0), "zero");
        advisorFund = _advisorFund;
    }
```

Listing 3.5: `NessToken::setMarketingFund()/setAdvisorFund()`

Note a number of routines in the `DarkNess Dollar` contracts can be similarly improved, including `Pool::setOracleDollar()/setOracleDark()/setOracleShare()/setOracleCollaterals()/setOracleCollateral()/setRedemptionDelay()/setTargetCollateralRatioConfig()/setTargetDarkOverShareRatioConfig()`, `CollateralReserve::setRouter()/setShareSellingPercent()/setMaxShareAmountToSell()/setShareToUsdcRouterPath()`, `Treasury::setTreasuryPolicy()/setOracleDollar()/setOracleDark()/setOracleShare()/setOracleCollaterals()/setOracleCollateral()/setCollateralReserve()/setProfitSharingFund()/setDarkInsuranceFund()/updateProtocol()`, and `TreasuryPolicy::setReserveShareState()`.

Recommendation Properly emit the related event when the above-mentioned functions are being invoked.

Status This issue has been fixed in this commit: `e6b3918`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `DarkNess Dollar` protocol, there are two privileged accounts, i.e., `owner and strategist`. These accounts play a critical role in governing and regulating the system-wide operations (e.g., add/remove a pool to/from the `Treasury`, pause/unpause minting/redeeming from an existing pool, update the `targetCollateralRatio_/targetDarkOverDarkShareRatio_` for an existing pool or update the whole protocol, and set the key parameters for the `DarkNess Dollar` protocol, etc.). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `Treasury` contract as an example and show the representative functions potentially affected by the privileges of the `owner/strategist` accounts.

```
    function setStrategistStatus(address _account, bool _status) external onlyOwner {
        strategist[_account] = _status;
        emit StrategistStatusUpdated(_account, _status);
    }
```

Listing 3.6: `Treasury::setStrategistStatus()`

```
    function setTreasuryPolicy(address _treasuryPolicy) public onlyOwner {
        require(_treasuryPolicy != address(0), "zero");
        treasuryPolicy = _treasuryPolicy;
    }

    function setOracleDollar(address _oracleDollar) external onlyOwner {
        require(_oracleDollar != address(0), "zero");
        oracleDollar = _oracleDollar;
    }

    function setOracleDark(address _oracleDark) external onlyOwner {
        require(_oracleDark != address(0), "zero");
        oracleDark = _oracleDark;
    }
```

```
    function setOracleShare(address _oracleShare) external onlyOwner {
        require(_oracleShare != address(0), "zero");
        oracleShare = _oracleShare;
    }
```

Listing 3.7: Treasury::setTreasuryPolicy()/setOracleDollar()/setOracleDark()/setOracleShare()

```
    function setOracleCollaterals(address[] memory _oracleCollaterals) external
        onlyOwner {
        require(_oracleCollaterals.length == 3, "invalid oracleCollaterals length");
        delete oracleCollaterals;
        for (uint256 i = 0; i < 3; i++) {
            oracleCollaterals.push(_oracleCollaterals[i]);
        }
    }

    function setOracleCollateral(uint256 _index, address _oracleCollateral) external
        onlyOwner {
        require(_oracleCollateral != address(0), "zero");
        oracleCollaterals[_index] = _oracleCollateral;
    }

    function setCollateralReserve(address _collateralReserve) public onlyOwner {
        require(_collateralReserve != address(0), "zero");
        collateralReserve_ = _collateralReserve;
    }
```

Listing 3.8: Treasury::setOracleCollaterals()/setOracleCollateral()/setCollateralReserve()

```
    function updateProtocol() external onlyStrategist {
        if (dollarPrice() > PRICE_PRECISION) {
            ITreasuryPolicy(treasuryPolicy).setMintingFee(20);
            ITreasuryPolicy(treasuryPolicy).setRedemptionFee(80);
        } else {
            ITreasuryPolicy(treasuryPolicy).setMintingFee(40);
            ITreasuryPolicy(treasuryPolicy).setRedemptionFee(40);
        }
        ...
    }
```

Listing 3.9: Treasury::updateProtocol()

```
    // Add new Pool
    function addPool(address pool_address) public onlyOwner {
        require(pools[pool_address] == false, "poolExisted");
        pools[pool_address] = true;
        pools_array.push(pool_address);
        emit PoolAdded(pool_address);
    }

    // Remove a pool
```

```
    function removePool(address pool_address) public onlyOwner {
        require(pools[pool_address] == true, "!pool");
        // Delete from the mapping
        delete pools[pool_address];
        // 'Delete' from the array by setting the address to 0x0
        for (uint256 i = 0; i < pools_array.length; i++) {
            if (pools_array[i] == pool_address) {
                pools_array[i] = address(0); // This will leave a null in the array and
                    keep the indices the same
                break;
            }
        }
        emit PoolRemoved(pool_address);
    }
```

Listing 3.10: `Treasury::addPool()/removePool()`

Note that if an existing pool is removed by the `owner`, the execution of the `redeem()/collectRedemption()` functions in the removed pool will revet, thus users may suffer asset losses.

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. 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. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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

Status This issue has been confirmed. The `DarkNess Dollar` team confirms that the `owner` will be transferred to timelock and later they will implement a `DAO` for Governance.

4 Conclusion

In this audit, we have analyzed the DarkNess Dollar design and implementation. DarkNess Dollar is a stablecoin with parts of its supply backed by collateral and parts of the supply algorithmic. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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

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

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

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