SMART CONTRACT AUDIT REPORT

for

DRT

Prepared By: Xiaomi Huang

PeckShield

May 5, 2023

Document Properties

Client	Cerchia
Title	Smart Contract Audit Report
Target	DRT
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 5, 2023	Xuxian Jiang	Final Release
1.0-rc	May 2, 2023	Xuxian Jiang	Release Candidate

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 DRT 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 6

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Accommodation of Non-ERC20-Compliant Tokens 11

3.2 Emit of Meaning Events Upon Important State Update 13

3.3 Trust Issue of Admin Keys 14

4 Conclusion **17**

References **18**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Direct Risk Transfer (DRT) 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 DRT

Direct Risk Transfer (DRT) is a bridge between those seeking protection from event risks and those who seek yield and are willing to provide protection. The former are the Buyers willing to buy protection by offering a Premium, and the latter are Sellers who sell protection to generate a yield by collecting the Premium. The size of the protection is called the Notional. Protection is against events based on an Index (e.g., an earthquake in a certain region) and further defined on standardized terms (e.g., the level of the earthquake and the protection term). Users can choose from a set of predefined event risks and standardized terms. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of DRT

Item	Description
Target	DRT
Website	https://www.cerchia.io/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 5, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this protocol assumes a trusted external oracle, which is not part of the audit.

- https://github.com/cerchia/DRT-Avalanche.git (1caaa52)

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

- https://github.com/cerchia/DRT-Avalanche.git (90aae5d)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 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) [7], 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 1.3: 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

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 DRT 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 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	1	
Low	2	
Informational	0	
Total	3	

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 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key DRT Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-002	Low	Emit of Meaning Events Upon Important State Update	Coding Practices	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Mitigated

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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 3.1: ZRX::transfer()/transferFrom()

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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `_processContingentSettlement()` routine in the `AnyAccountOperationsFacet` contract. If the `USDT` token is supported as `_token`, the unsafe version of `token.transfer(initiator, fundsToReturn)` (line 319) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
289     function _processContingentSettlement(
290         address callerAddress,
291         uint64 date,
292         LibStructStorage.Deal storage deal
293     ) private {
294         Storage.CerchiaDRTStorage storage s = Storage.getStorage();
295 
296         if (deal.state == LibStructStorage.DealState.BidLive  deal.state ==
            LibStructStorage.DealState.AskLive) {
297             // If deal is BidLive or AskLive, and we are past expiration date, expire
                deal and send back funds to initiator
298             // Otherwise, there is nothing to be done
299             if (date >= deal.expiryDate) {
300                 uint256 dealId = deal.id;
301                 address initiator = deal.initiator;
302                 LibStructStorage.DealState dealState = deal.state;
303                 IERC20 token = IERC20(deal.voucher.token);
304                 uint128 fundsToReturn = deal.funds;
305 
306                 // No need to check if deal exists, reaching this point means it exists
307                 // Delete expired deal
308                 s._dealsSet.deleteById(dealId);
309 
310                 // Emit the correct event
311                 if (dealState == LibStructStorage.DealState.BidLive) {
312                     emit BidLiveDealExpired(dealId, initiator, fundsToReturn);
313                 } else {
314                     emit AskLiveDealExpired(dealId, initiator, fundsToReturn);
```

```
                }

                // Transfer funds back to initiator of deal, since BidLive/AskLive
                // implies either buyer of seller are involved, but not both
                require(token.transfer(initiator, fundsToReturn), LibStructStorage.
                    TOKEN_TRANSFER_FAILED);
            }
        } ...
    }
```

Listing 3.2: AnyAccountOperationsFacet::_processContingentSettlement()

The same issue is also applicable to a number of other routines, including _processEoDForLiveDeal(), claimBack(), userUpdateDealFromBidToMatched(), userUpdateDealFromAskToMatched(), and userCreateNewDealAsBid()().

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

Status The issue has been fixed by this commit: 0f43a90.

3.2 Emit of Meaning Events Upon Important State Update

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: QoreAdmin
- Category: Coding Practices [5]
- 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 AccessControlFacet contract as an example. While examining the events that reflect the protocol dynamics, we notice the related events are not emitted when various important parameters are updated. To elaborate, we show below related two routines, i.e., _setFeeAddress() and _setOracleAddress(). The changes to respective states _feeAddress and _oracleAddress are sensitive, which brings the need to emit meaningful events to reflect their changes.

```
    function _setFeeAddress(address feeAddress) private {
        require(
```

```
            !hasRole(LibStructStorage.OWNER_ROLE, feeAddress),
            LibStructStorage.ACCOUNT_TO_BE_FEE_ADDRESS_IS_ALREADY_OWNER
        );
        require(
            !hasRole(LibStructStorage.OPERATOR_ROLE, feeAddress),
            LibStructStorage.ACCOUNT_TO_BE_FEE_ADDRESS_IS_ALREADY_OPERATOR
        );

        LibAccessControlStorage.getStorage()._feeAddress = feeAddress;
    }

    /**
     * @dev Sets the oracle address. Address should not already be owner or operator
     */
    function _setOracleAddress(address oracleAddress) private {
        LibAccessControlStorage.getStorage()._oracleAddress = oracleAddress;
    }
```

Listing 3.3: `AccessControlFacet::_setFeeAddress()/_setOracleAddress()`

Recommendation Properly emit the related events in all updates on sensitive states or configurations. They are very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: `0f43a90`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `DRT` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, assign other roles, as well as upgrade the proxy). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function ownerAddNewStandard(
        bytes32 configurationId,
        string calldata symbol,
        uint64 startDate,
        uint64 maturityDate,
        uint128 feeInBps,
        int128 strike,
```

```
        uint8 exponentOfTenMultiplierForStrike
    ) external onlyOwner isNotDeactivatedForOwner isExactDate(startDate) isEndOfDate(
        maturityDate) {
        ...
    }

    /**
     * @inheritdoc IOwnerOperations
     * @dev  On Avalanche Fuji Chain, owner would call this with ("USDC", <
         REAL_USDC_ADDRESS_ON_FUJI>)
     */
    function ownerAddNewToken(string calldata denomination, address token) external
        onlyOwner isNotDeactivatedForOwner {
        ...
    }

    /**
     * @inheritdoc  IOwnerOperations
     * @dev          Since we expect a small number of standards, controller by owners,
         the below "for" loops are
     *               not a gas or DoS concern
     */
    function ownerDeleteStandards(string[] calldata symbols) external onlyOwner
        isNotDeactivatedForOwner {
        ...
    }

    /**
     * @inheritdoc IOwnerOperations
     * @dev      Since we expect a small number of tokens, controller by owners, the
         below "for" loops are
     *           not a gas or DoS concern
     */
    function ownerDeleteTokens(string[] calldata symbols) external onlyOwner
        isNotDeactivatedForOwner {
        ...
    }
```

Listing 3.4: Example Privileged Operations in OwnerOperationsFacet

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Moreover, it should be noted that current contracts are to be deployed behind a proxy with the typical Diamond implementation. And naturally, there is a need to properly manage the admin privileges as they are capable of upgrading the entire protocol implementation.

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 by the team. The team intends to have owner account for deployment and adding items like standards and tokens, and these accounts are protected in AWS, for which access is extremely limited, with 2FA.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Direct Risk Transfer` (DRT) protocol, which is a bridge between those seeking protection from event risks and those who seek yield and are willing to provide protection. The former are the `Buyers` willing to buy protection by offering a `Premium`, and the latter are `Sellers` who sell protection to generate a yield by collecting the `Premium`. The size of the protection is called the `Notional`. Protection is against events based on an `Index` (e.g., an earthquake in a certain region) and further defined on standardized terms (e.g., the level of the earthquake and the protection term). Users can choose from a set of predefined event risks and standardized terms. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

[1] 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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