SMART CONTRACT AUDIT REPORT

for

Duet

Prepared By: Yiqun Chen

PeckShield

January 29, 2022

Document Properties

Client	Duet Finance
Title	Smart Contract Audit Report
Target	Duet
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 29, 2022	Xiaotao Wu	Final Release
1.0-rc	January 28, 2022	Xiaotao Wu	Release Candidate #1

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 source code of the `Duet` 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 Duet

`Duet` is a multi-chain synthetic asset protocol with a hybrid mechanism (overcollateralization + algorithm-pegged) that sharpens assets to be traded on the blockchain. A duet in music refers to a piece of music where two people play different parts or melodies. Similarly, the `Duet` protocol allows traders to replicate the real-world tradable assets in a decentralized finance ecosystem.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Duet

Item	Description
Name	Duet Finance
Website	https://duet.finance/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 29, 2022

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

- https://github.com/duet-protocol/Duet-Over-Collateralization-us.git (ffd1a9a)

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

- https://github.com/duet-protocol/duet-collateral-contracts.git (92452da)

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

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 accordingly classified into four categories, 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

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

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. 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 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 design and implementation of the `Duet` protocol 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	4	■ ■ ■ ■
Informational	1	■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly DYToken From Improper Pool Initialization	Time and State	Fixed
PVE-002	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-003	Low	Lack of BNB Handling In DYTokenBase::inCaseTokensGetStuck()	Business Logics	Fixed
PVE-004	Low	Possible Sandwich/MEV Attacks In Duet	Time and State	Confirmed
PVE-005	Low	Potential Lockup Of Tokens Leftover In DuetZap::tokenToLp()	Business Logics	Confirmed
PVE-006	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 Possible Costly DYToken From Improper Pool Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: DYTokenERC20/DYTokenNative
- Category: Time and State [8]
- CWE subcategory: CWE-362 [2]

Description

The DYTokenERC20 contract of the Duet protocol provides a public depositTo() function for users to deposit the underlying token to the DYToken contract and mint the corresponding shares of DYToken to the users. While examining the DYToken share calculation with the given underlying token amount, we notice an issue that may unnecessarily make the underlying token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the depositTo() routine. The issue occurs when the deposit pool is being initialized under the assumption that the current pool is empty.

```
    function depositTo(address _to, uint _amount, address _toVault) public override {
        uint total = underlyingTotal();
        IERC20 underlyingToken = IERC20(underlying);

        uint before = underlyingToken.balanceOf(address(this));
        underlyingToken.safeTransferFrom(msg.sender, address(this), _amount);
        uint realAmount = underlyingToken.balanceOf(address(this)) - before; //
            Additional check for deflationary tokens
        require(realAmount >= _amount, "illegal amount");

        uint shares = 0;
        if (totalSupply() == 0) {
          shares = _amount;
        } else {
          shares = _amount * totalSupply() / total;
        }
```

```
44
45          //
46          if(_toVault != address(0)) {
47            require(_toVault == IController(controller).dyTokenVaults(address(this)), "
                mismatch dToken vault");
48            _mint(_toVault, shares);
49            IDepositVault(_toVault).syncDeposit(address(this), shares, _to);
50          } else {
51            _mint(_to, shares);
52          }
53
54          earn();
55        }
```

Listing 3.1: `DYTokenERC20::depositTo()`

Specifically, when the `deposit` pool is being initialized, the `shares` value directly takes the value of `_amount` (line 40), which is manipulatable by the malicious actor. As this is the first time to deposit, the `totalSupply()` equals the given input amount, i.e., `_amount = 1 WEI`. With that, the actor can further donate a huge amount of `underlying` to `DYTokenERC20` contract with the goal of making the `DYToken` extremely expensive.

An extremely expensive `DYToken` can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed `shares` for deposited assets (line 42). If truncated to be zero, the deposited assets are essentially considered dust and kept by the contract without returning any `DYToken`.

Note the `DYTokenNative::depositCoin()/depositTo()` routines share a similar issue.

Recommendation Revise current execution logic of above mentioned functions to defensively calculate the mint amount when the `deposit pool` is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been fixed in the following commit: `e6f1a47`.

3.2 Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple contracts
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [3]

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 FeeConf contract as an example. While examining the event that reflect the FeeConf dynamics, we notice there is a lack of emitting related event to reflect important state change. Specifically, when the setConfig() is being called, there is no corresponding event being emitted to reflect the occurrence of setConfig().

```
    function setConfig(bytes32 _key, address _receiver, uint16 _rate) public onlyOwner {
        require(_receiver != address(0), "INVALID_RECEIVE");
        ReceiverRate storage conf = configs[_key];
        conf.receiver = _receiver;
        conf.rate = _rate;
    }
```

Listing 3.2: FeeConf::setConfig

Note a number of routines in the Duet protocol contracts can be similarly improved, including DYTokenBase::setController(), DuetZap::setRoutePairAddress(), AppController::setVaultStates(), BaseStrategy::setMinHarvestAmount()/setController()/setFeeConf(), and StrategyForPancakeLP::setToken0Path()/setToken2Path().

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

Status This issue has been fixed in the following commit: e169a53.

3.3 Lack of BNB Handling In DYTokenBase::inCaseTokensGetStuck()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: DYTokenBase
- Category: Business Logics [10]
- CWE subcategory: CWE-708 [5]

Description

The DYTokenBase contract provides the inCaseTokensGetStuck() function for the owner to withdraw the ERC20 tokens from the contract in case these tokens got stuck. The DYTokenBase contract can also receive BNB via the depositCoin() function which is defined as payable. However, the current implementation logic of the inCaseTokensGetStuck() function only considers the case of ERC20 tokens. Therefore, the owner can not recover BNB if there are BNBs got stuck in the contract.

```
function inCaseTokensGetStuck(address _token, uint _amount) public onlyOwner {
    IERC20(_token).transfer(owner(), _amount);
}
```

Listing 3.3: DYTokenBase::inCaseTokensGetStuck()

Recommendation Consider the scenario that BNB may also got stuck in the contract.

Status This issue has been fixed. The Duet team has removed the inCaseTokensGetStuck() function from the DYTokenBase contract.

3.4 Possible Sandwich/MEV Attacks In Duet

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [11]
- CWE subcategory: CWE-682 [4]

Description

The DuetZap contract has a helper routine, i.e., _swap(), that is designed to swap one token for another. It has a rather straightforward logic in allowing router to transfer the funds by calling swapExactTokensForTokens() to actually perform the intended token swap.

```
    function _swap(address _from, uint amount, address _to, address receiver) private
        returns (uint) {
        address intermediate = routePairAddresses[_from];
        if (intermediate == address(0)) {
            intermediate = routePairAddresses[_to];
        }

        address[] memory path;

        if (intermediate == address(0)  _from == intermediate  _to == intermediate ) {
            // [DUET, BUSD] or [BUSD, DUET]
            path = new address[](2);
            path[0] = _from;
            path[1] = _to;
        } else {
            path = new address[](3);
            path[0] = _from;
            path[1] = intermediate;
            path[2] = _to;
        }

        uint[] memory amounts = router.swapExactTokensForTokens(amount, 0, path,
            receiver, block.timestamp);
        return amounts[amounts.length - 1];
    }
```

Listing 3.4: `DuetZap::_swap()`

To elaborate, we show above the `_swap()` routine. We notice the token swap is routed to `router` and the actual swap operation `swapExactTokensForTokens()` essentially does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Note the `DuetZap::_swapTokenForBNB()/_swapBNBForToken()/_swapBNBToLp()/zapOut()` and `StrategyForPancakeLP::doHarvest()` routines share a similar issue.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed. And the team clarifies that, `MEV` attacks are acceptable for the above mentioned scenarios.

3.5 Potential Lockup Of Tokens Leftover In DuetZap::tokenToLp()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DuetZap`
- Category: Business Logics [10]
- CWE subcategory: CWE-754 [6]

Description

In the `DuetZap` contract, the `tokenToLp()` function is designed to get `UniswapV2 LP` tokens via a single `ERC20` asset. While examining its logics, we notice there may have leftover tokens locked in the `DuetZap` contract.

To elaborate, we show below the related code snippet of the `DuetZap` contract. In the `tokenToLp()` function, it comes to our attention with the following action sequences: The `_swap()` function is firstly called (line 65) to swap half the number of the `_token` (specified by the function input parameter) to `other` and then the `addLiquidity()` function is called (line 68) to add the remaining half of the `_token` and the exchanged `other` to the `UniswapV2 _token + other pair` to provide liquidity. This is reasonable under the assumption that those tokens approved to the `UniswapV2 router` contract happen to be used entirely to provide liquidity. Otherwise, the leftover tokens will be locked in the contract. We suggest to calculate the actual amount of tokens before transferring them into the contract.

```
1755    function tokenToLp(address _token, uint amount, address _lp, bool needDeposit)
            external {
1756        address receiver = msg.sender;
1757        if (needDeposit) {
1758          receiver = address(this);
1759        }
1760        IERC20Upgradeable(_token).safeTransferFrom(msg.sender, address(this), amount);
1761        _approveTokenIfNeeded(_token, address(router));
1762
1763        IPair pair = IPair(_lp);
1764        address token0 = pair.token0();
1765        address token1 = pair.token1();
1766
1767        uint liquidity;
1768
1769        if (_token == token0  _token == token1) {
1770            // swap half amount for other
1771            address other = _token == token0 ? token1 : token0;
1772            _approveTokenIfNeeded(other, address(router));
```

```
            uint sellAmount = amount / 2;

            uint otherAmount = _swap(_token, sellAmount, other, address(this));
            pair.skim(address(this));

            (, , liquidity) = router.addLiquidity(_token, other, amount - sellAmount,
                otherAmount, 0, 0, receiver, block.timestamp);
        } else {
            uint bnbAmount = _token == wbnb ? _safeSwapToBNB(amount) : _swapTokenForBNB(
                _token, amount, address(this));
            liquidity = _swapBNBToLp(_lp, bnbAmount, receiver);
        }

        emit ZapToLP(_token, amount, _lp, liquidity);
        if (needDeposit) {
          deposit(_lp, liquidity, msg.sender);
        }

    }
```

Listing 3.5: `DuetZap::tokenToLp()`

Note the `_swapBNBToLp()` routine in the same contract can be similarly improved.

Recommendation Add additional handling logic for the above mentioned functions to return the leftover assets to the user (if any).

Status The issue has been confirmed.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [1]

Description

In the `Duet` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., mint/burn `Duet` tokens, withdraw assets from `Duet` contracts, set the key parameters, etc.).

In the following, we use the `Duet` contract as an example and show the representative functions potentially affected by the privilege of the `owner` account. The `owner` is privileged to mint more `Duet` tokens into circulation or burn `Duet` tokens from circulation.

```
    function mint(address account, uint256 amount) public onlyOwner {
        _mint(account, amount);
    }

    function burn(address account, uint256 amount) public onlyOwner {
        _burn(account, amount);
    }
```

Listing 3.6: `Duet::mint()/burn()`

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

Recommendation Make the list of extra privileges granted to `owner` explicit to `Duet` protocol users.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Duet` protocol. `Duet` is a multi-chain synthetic asset protocol with a hybrid mechanism (overcollateralization + algorithm-pegged) that sharpens assets to be traded on the blockchain. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[5] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

[6] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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