SMART CONTRACT AUDIT REPORT

for

Duet Bond

Prepared By: Patrick Lou

PeckShield
March 19, 2022

Document Properties

Client	Duet Finance
Title	Smart Contract Audit Report
Target	Duet Bond
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 19, 2022	Xiaotao Wu	Final Release
1.0-rc	March 18, 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 Duet Bond feature, 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 Duet Bond

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 audited Duet Bond feature allows for the protocol administrator to issue bonds through the bond factory and set the reward pools. The DYToken deposited to the Duet Vault by a user will be synchronized to the reward pool and get Epoch tokens in return. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Duet Bond

Item	Description
Name	Duet Finance
Website	https://duet.finance/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 19, 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-bond-contract.git (549f7d3)

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-bond-contract.git (0e841c5)

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 the 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.

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

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

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

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

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

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Duet Bond` smart contract. 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	1	
Medium	2	
Low	2	
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 high-severity vulnerability, 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Duet Bond Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Exposure Of Permissioned VaultFarm::massUpdatePools()	Business Logic	Resolved
PVE-002	Medium	Incorrect Epoch Removal Logic In VaultFarm::removePoolEpoch	Business Logic	Resolved
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-004	Informational	Improved Sanity Checks Of System/-Function Parameters	Coding Practices	Resolved
PVE-005	Low	Incorrect start Update Logic In SingleBond::renewal()	Business Logic	Resolved
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 Exposure Of Permissioned VaultFarm::massUpdatePools()

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

Description

The `Duet Bond` feature has the `VaultFarm` contract to farm the `Epoch` tokens for vault users. When examining the implementation of the `VaultFarm` contract, we notice the presence of a specific routine, i.e., `massUpdatePools()`. As the name indicates, this routine is used to update reward variables for all pools with the given input parameters. To elaborate, we show below the code snippet of this function.

```
    function massUpdatePools(address[] memory epochs, uint256[] memory rewards) public {
        uint256 poolLen = pools.length;
        uint256 epochLen = epochs.length;


        uint[] memory epochArr = new uint[](epochLen);
        for (uint256 pi = 0; pi < poolLen; pi++) {
          for (uint256 ei = 0; ei < epochLen; ei++) {
            epochArr[ei] = rewards[ei] * allocPoint[pools[pi]] / totalAllocPoint;
          }
          Pool(pools[pi]).updateReward(epochs, epochArr, periodFinish);
        }

        epochRewards = rewards;
        lastUpdateSecond = block.timestamp;
    }
```

Listing 3.1: `VaultFarm::massUpdatePools()`

However, we notice that this routine is currently permissionless, which means it can be invoked by anyone to update reward variables for all pools according to his wish. To fix, the function type needs to be changed from `public` to `internal` such that this function can only be accessed internally.

Recommendation Adjust the function type from `public` to `internal` for the above `massUpdatePools()` function.

Status This issue has been fixed in the following commit: `655a706`.

3.2 Incorrect Epoch Removal Logic In VaultFarm::removePoolEpoch

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `VaultFarm`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `VaultFarm` contract provides an external `removePoolEpoch()` function for the privileged `Owner` account to remove a specified `epoch` token from a specified pool. Our analysis with this routine shows its current logic is not correct.

To elaborate, we show below its code snippet. It comes to our attention that there is a lack of pending rewards handling and related storage arrays `epochs/epochRewards` updates before removing an `epoch` token from a pool. If the storage arrays `epochs/epochRewards` are not updated timely, this removed `epoch` token will be added to the pool again if the `newPool()/updatePool()/appendReward()` functions are called by the privileged `Owner` account.

```
    function removePoolEpoch(address pool, address epoch) external onlyOwner {
        Pool(pool).remove(epoch);
    }
```

Listing 3.2: `VaultFarm::removePoolEpoch()`

```
    // remove some item for saving gas (array issue).
    // should only used when no such epoch assets.
    function remove(address epoch) external onlyFarming {
        require(validEpoches[epoch], "Not a valid epoch");
        validEpoches[epoch] = false;

        uint len = epoches.length;
        for (uint i = 0; i < len; i++) {
```

```
            if( epoch == epoches[i]) {
                if (i == len - 1) {
                    epoches.pop();
                    break;
                } else {
                  epoches[i] = epoches[len - 1];
                  epoches.pop();
                  break;
                }
            }
        }
    }
```

Listing 3.3: `Pool::remove()`

Recommendation Add pending rewards handling and storage arrays `epoches/epochRewards` updates logic before removing an `epoch` token from a pool.

Status This issue has been fixed in the following commit: `0e841c5`.

3.3 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SingleBond/SingleBondsFactory`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

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

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
```

```
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

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

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: `USDT Token Contract`

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

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

In the following, we use the `SingleBondsFactory::renewal()` routine as an example. If the `USDT` token is supported as `rewardtoken`, the unsafe version of `token.approve(address(bondAddr), totalAmount)` may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
    function renewal (SingleBond bondAddr, uint256 _phasenum,uint256 _principal,uint256
        _interestone) external onlyOwner {
        IERC20 token = IERC20(bondAddr.rewardtoken());
        uint totalAmount = _phasenum * _interestone + _principal;
        require(token.balanceOf(msg.sender)>= totalAmount, "factory:no balance");
        token.safeTransferFrom(msg.sender, address(this), totalAmount);
        token.approve(address(bondAddr), totalAmount);

        bondAddr.renewal(_phasenum, _principal, _interestone);
    }
```

Listing 3.5: `SingleBondsFactory::renewal()`

Note that a number of routines can be similarly improved, including `SingleBondsFactory::newBonds()/renewSingleEpoch()`, and `SingleBond::initBond()/renewal()/renewSingleEpoch()`.

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

Status This issue has been fixed in the following commit: `655a706`.

3.4 Improved Sanity Checks Of System/Function Parameters

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `VaultFarm`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In the `VaultFarm` contract, the `newReward()` function allows for the privileged `Owner` account to add `Epoch` tokens as rewards for existing pools. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the full implementation of the `newReward()` function. Specifically, there is a lack of length verification for the input parameters. Thus the execution of `IERC20(epochs[i]).transferFrom(msg.sender, address(this), rewards[i])` will revert if `epochs.length > rewards.length` (line 139).

```
129     function newReward(address[] memory epochs, uint256[] memory rewards, uint duration)
             public onlyOwner {
130         require(block.timestamp >= periodFinish, 'period not finish');
131         require(duration > 0, 'duration zero');
132
133         periodFinish = block.timestamp + duration;
134         epoches = epochs;
135         massUpdatePools(epochs, rewards);
136
137         for (uint i = 0 ; i < epochs.length; i++) {
138           require(IEpoch(epochs[i]).bond() == bond, "invalid epoch");
139           IERC20(epochs[i]).transferFrom(msg.sender, address(this), rewards[i]);
140         }
141     }
```

Listing 3.6: `VaultFarm::newReward()`

Recommendation Add length verification for the input parameters.

Status This issue has been fixed in the following commit: `655a706`.

3.5 Incorrect start Update Logic In SingleBond::renewal()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SingleBond`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `SingleBond` contract provides an external `renewal()` function for the privileged `Owner` (i.e., the `SingleBondsFactory` contract) to create `Epoch` token contracts and transfer the `rewardtoken` to these `Epoch` token contracts as the underlying tokens.

In the following, we show the code snippet of the `renewal()` routine. Our analysis with this routine shows that the update of the state variables `newstart/start` is not implemented correctly. Specifically, the conditions for updating the state variables `newstart/start` should be `block.timestamp > end`, instead of current `block.timestamp + duration >= end` (line 76).

```
71      //renewal bond will start at next phase
72      function renewal (uint256 _phasenum,uint256 _principal,uint256 _interestone)
            external onlyOwner {
73          uint256 needcreate = 0;
74          uint256 newstart = end;
75          uint256 renewphase = (block.timestamp - start)/duration + 1;
76          if(block.timestamp + duration >= end){
77              needcreate = _phasenum;
78              newstart = block.timestamp;
79              start = block.timestamp;
80              phasenum = 0;
81          }else{
82              if(block.timestamp + duration*_phasenum <= end) {
83                  needcreate = 0;
84              } else {
85                  needcreate = _phasenum - (end - block.timestamp)/duration;
86              }
87          }
88
89          ...
90      }
```

Listing 3.7: `SingleBond::renewal()`

Recommendation Correct the above `renewal()` logic by fixing the `if` statement as follows.

```
    //renewal bond will start at next phase
    function renewal (uint256 _phasenum,uint256 _principal,uint256 _interestone)
        external onlyOwner {
        uint256 needcreate = 0;
        uint256 newstart = end;
        uint256 renewphase = (block.timestamp - start)/duration + 1;
        if(block.timestamp > end){
            needcreate = _phasenum;
            newstart = block.timestamp;
            start = block.timestamp;
            phasenum = 0;
        }else{
            if(block.timestamp + duration*_phasenum <= end) {
                needcreate = 0;
            } else {
                needcreate = _phasenum - (end - block.timestamp)/duration;
            }
        }

        ...
    }
```

Listing 3.8: `SingleBond::renewal()`

Status This issue has been fixed in the following commit: `655a706`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Duet Bond` feature, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., set pool implementation, approve vault, create rewards or append rewards for existing pools, remove `Epoch` from an existing pool, create/update pool, emergency withdraw `Epoch` tokens from the `VaultFarm` contract, renew `bond`, and renew single `epoch` for an existing `bond`, set `period` for the `DexUSDOracle` contract, etc.). It also has the privilege to control or govern the flow of assets managed by this feature. 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.

```
    function setPoolImp(address _poolImp) external onlyOwner {
        poolImp = _poolImp;
    }

    function approveVault(address vault, bool approved)  external onlyOwner {
        vaults[vault] = approved;
        emit VaultApproved(vault, approved);
    }
```

Listing 3.9: `VaultFarm::setPoolImp()/approveVault()`

```
    // epochs need small for gas issue.
    function newReward(address[] memory epochs, uint256[] memory rewards, uint duration)
         public onlyOwner {
        require(block.timestamp >= periodFinish, 'period not finish');
        require(duration > 0, 'duration zero');

        periodFinish = block.timestamp + duration;
        epoches = epochs;
        massUpdatePools(epochs, rewards);

        for (uint i = 0 ; i < epochs.length; i++) {
            require(IEpoch(epochs[i]).bond() == bond, "invalid epoch");
            IERC20(epochs[i]).transferFrom(msg.sender, address(this), rewards[i]);
        }
    }

    function appendReward(address epoch, uint256 reward) public onlyOwner {
        require(block.timestamp < periodFinish, 'period not finish');
        require(IEpoch(epoch).bond() == bond, "invalid epoch");

        bool inEpoch;
        uint i;
        for (; i < epoches.length; i++) {
            if (epoch == epoches[i]) {
                inEpoch = true;
                break;
            }
        }

        uint[] memory leftRewards = calLeftAwards();
        if (!inEpoch) {
            epoches.push(epoch);
            uint[] memory newleftRewards = new uint[](epoches.length);
            for (uint j = 0; j < leftRewards.length; j++) {
                newleftRewards[j] = leftRewards[j];
        }
        newleftRewards[leftRewards.length] = reward;

        massUpdatePools(epoches, newleftRewards);
        } else {
            leftRewards[i] += reward;
            massUpdatePools(epoches, leftRewards);
```

```
        }

        IERC20(epoch).transferFrom(msg.sender, address(this), reward);
    }

    function removePoolEpoch(address pool, address epoch) external onlyOwner {
        Pool(pool).remove(epoch);
    }
```

Listing 3.10: `VaultFarm::newReward()/appendReward()/removePoolEpoch()`

```
    function newPool(uint256 _allocPoint, address asset) public onlyOwner {
        require(assetPool[asset] == address(0), "pool exist!");

        address pool = createClone(poolImp);
        Pool(pool).init();

        pools.push(pool);
        allocPoint[pool] = _allocPoint;
        assetPool[asset] = pool;
        totalAllocPoint = totalAllocPoint + _allocPoint;

        emit NewPool(asset, pool);
        uint[] memory leftRewards = calLeftAwards();
        massUpdatePools(epoches,leftRewards);
    }

    function updatePool(uint256 _allocPoint, address asset) public onlyOwner {
        address pool = assetPool[asset];
        require(pool != address(0), "pool not exist!");

        totalAllocPoint = totalAllocPoint - allocPoint[pool] + _allocPoint;
        allocPoint[pool] = _allocPoint;

        uint[] memory leftRewards = calLeftAwards();
        massUpdatePools(epoches,leftRewards);
  }
```

Listing 3.11: `VaultFarm::newPool()/updatePool()`

```
    function emergencyWithdraw(address[] memory epochs, uint256[] memory amounts)
        external onlyOwner {
        require(epochs.length == amounts.length, "mismatch length");
        for (uint i = 0 ; i < epochs.length; i++) {
        IERC20(epochs[i]).transfer(msg.sender, amounts[i]);
        }
    }
```

Listing 3.12: `VaultFarm::emergencyWithdraw()`

```
    function renewal (SingleBond bondAddr, uint256 _phasenum,uint256 _principal,uint256
        _interestone) external onlyOwner {
```

```
        IERC20 token = IERC20(bondAddr.rewardtoken());
        uint totalAmount = _phasenum * _interestone + _principal;
        require(token.balanceOf(msg.sender)>= totalAmount, "factory:no balance");
        token.safeTransferFrom(msg.sender, address(this), totalAmount);
        token.approve(address(bondAddr), totalAmount);

        bondAddr.renewal(_phasenum, _principal, _interestone);
    }

    function renewSingleEpoch(SingleBond bondAddr, uint256 id, uint256 amount, address
        to) external onlyOwner{
        IERC20 token = IERC20(bondAddr.rewardtoken());
        token.safeTransferFrom(msg.sender, address(this), amount);
        token.approve(address(bondAddr), amount);
        bondAddr.renewSingleEpoch(id,amount,to);
    }
```

Listing 3.13: `SingleBondsFactory::renewal()/renewSingleEpoch()`

```
    function setPeriod(uint _period) external onlyOwner {
        period = _period;
        emit PeriodChanged(_period);
    }
```

Listing 3.14: `DexUSDOracle::setPeriod()`

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.

4 Conclusion

In this audit, we have analyzed the `Duet Bond` design and implementation. The audited `Duet Bond` feature allows for the protocol administrator to issue bonds through the bond factory and set the reward pools. The `DYToken` deposited to the `Duet Vault` by a user will be synchronized to the reward pool and users will get `Epoch` tokens in return. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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