SMART CONTRACT AUDIT REPORT

for

DeFi Yield Protocol v2

Prepared By: Yiqun Chen

PeckShield
November 08, 2021

Document Properties

Client	DeFi Yield Protocol
Title	Smart Contract Audit Report
Target	DeFi Yield Protocol v2
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 08, 2021	Xuxian Jiang	Final Release
1.0-rc	November 4, 2021	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related source code of the `DYP v2` 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 DYP v2

The `DeFi Yield Protocol (DYP)` (V2) protocol is a unique platform that allows anyone to provide liquidity and to be rewarded on `Ethereum`. At the same time, the platform maintains both token price stability as well as secure and simplified DeFi for end users by integrating a DYP anti-manipulation feature. In order to lower the risk of `DYP` price volatility, all pool rewards are automatically converted from `DYP` to ETH by the smart contract, and the ETH is distributed as a reward to the liquidity providers.

The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	DeFi Yield Protocol
Website	https://dyp.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 08, 2021

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

- https://github.com/dypfinance/Buyback-Farm-Stake-Governance-V2 (81c9321)

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

- https://github.com/dypfinance/Buyback-Farm-Stake-Governance-V2 (ddb3882)

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 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 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

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) [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.

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 implementation of the DYP 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	0	
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 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 4 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of DeFi Yield Protocol v2 Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-002	Low	Improved Logic of deposit() When addLiquidityAndGetAmountToDeposit()	Business Logic	Confirmed
PVE-003	Low	Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Business Logic	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Validation Of Function Arguments

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: FarmProRata
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

In the FarmProRata contract, the addLiquidityAndGetAmountToDeposit() routine (see the code snippet below) is provided to add _rewardTokenReceived amount of tokenA and _baseTokenReceived amount of tokenB into the pool as liquidity via the uniswapRouterV2::addLiquidity() routine. To elaborate, we show below the related code snippet.

```
    function addLiquidityAndGetAmountToDeposit(
        uint _rewardTokenReceived,
        uint _baseTokenReceived,
        uint[] memory minAmounts,
        uint _deadline
    ) private returns (uint) {
        uint oldLpBalance = IERC20(trustedDepositTokenAddress).balanceOf(address(this));

        IERC20(trustedRewardTokenAddress).safeApprove(address(uniswapRouterV2), 0);
        IERC20(trustedRewardTokenAddress).safeApprove(address(uniswapRouterV2),
            _rewardTokenReceived);

        IERC20(trustedBaseTokenAddress).safeApprove(address(uniswapRouterV2), 0);
        IERC20(trustedBaseTokenAddress).safeApprove(address(uniswapRouterV2),
            _baseTokenReceived);

        uniswapRouterV2.addLiquidity(
            trustedRewardTokenAddress,
            trustedBaseTokenAddress,
            _rewardTokenReceived,
            _baseTokenReceived,
```

```
            /*amountLiquidityMin_rewardTokenReceived*/minAmounts[2],
            /*amountLiquidityMin_baseTokenReceived*/minAmounts[3],
            address(this),
            _deadline
        );

        uint newLpBalance = IERC20(trustedDepositTokenAddress).balanceOf(address(this));
        uint lpTokensReceived = newLpBalance.sub(oldLpBalance);

        return lpTokensReceived;
    }
```

Listing 3.1: `FarmProRata::addLiquidityAndGetAmountToDeposit()`

It comes to our attention that the `Uniswap V2 Router` has implicit assumptions on the `_addLiquidity()` routine, which is internally called by the `uniswapRouterV2::addLiquidity()` routine. To elaborate, we show below the related code snippet.

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory, tokenA,
            tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'UniswapV2Router:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.2: `UniswapV2Router02::_addLiquidity()`

The above routine takes two amounts: `amountXDesired` and `amountXMin`. `amountXDesired` determines the desired amount for adding liquidity to the pool and `amountXMin` determines the minimum amount of assets used. There is an implicit condition required: `amountADesired >= amountAMin`, and `amountBDesired >= amountBMin`. Otherwise, the `amountXMin` may not trigger reverts and are simply ignored because the code above are performing asymmetric checks for the amounts. Hence, without stating these assumptions, slippage control for some trades on `Uniswap V2 Router` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `_rewardTokenReceived >= minAmounts[2]` and `_baseTokenReceived >= minAmounts[3]` explicitly in the `addLiquidityAndGetAmountToDeposit()` function.

Status The issue has been fixed by this commit: `ddb3882`.

3.2 Improved Logic of deposit() When addLiquidityAndGetAmountToDeposit()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FarmProRata`
- Category: Business Logics [8]
- CWE subcategory: CWE-754 [4]

Description

According to the `FarmProRata` design, the `deposit()` routine will convert user assets into the demanding tokens and add liquidity for the target `LP` tokens (via `addLiquidityAndGetAmountToDeposit()`). While examining the process of adding liquidity, we notice certain assets may be left in the `FarmProRata` contract.

To elaborate, we show below the related code snippet of the `FarmProRata()` contract.

```
    function deposit(
        address depositToken,
        uint amountToStake,
        uint[] memory minAmounts,
        // uint _amountOutMin_25Percent, // 0
        // uint _amountOutMin_stakingReferralFee, // 1
        // uint amountLiquidityMin_rewardTokenReceived, // 2
        // uint amountLiquidityMin_baseTokenReceived, // 3
        // uint _amountOutMin_rewardTokenReceived, // 4
        // uint _amountOutMin_baseTokenReceived, // 5
        // uint _amountOutMin_claimAsToken_dyp, // 6
        // uint _amountOutMin_attemptSwap, // 7
        uint _deadline
```

```
    ) public noContractsAllowed notDuringEmergency {
        require(minAmounts.length == 8, "Invalid minAmounts length!");

        require(trustedClaimableTokens[depositToken], "Invalid deposit token!");

        // can deposit reward token directly
        // require(depositToken != trustedRewardTokenAddress, "Cannot deposit reward
            token!");

        require(depositToken != trustedDepositTokenAddress, "Cannot deposit LP directly!
            ");
        require(depositToken != address(0), "Deposit token cannot be 0!");

        require(amountToStake > 0, "Invalid amount to Stake!");

        IERC20(depositToken).safeTransferFrom(msg.sender, address(this), amountToStake);

        uint fee = amountToStake.mul(STAKING_FEE_RATE_X_100).div(100e2);
        uint amountAfterFee = amountToStake.sub(fee);
        if (fee > 0) {
            IERC20(depositToken).safeTransfer(feeRecipientAddress, fee);
        }

        uint _75Percent = amountAfterFee.mul(75e2).div(100e2);
        uint _25Percent = amountAfterFee.sub(_75Percent);

        uint amountToDepositByContract = doSwap(depositToken,
            trustedPlatformTokenAddress, _25Percent, /*_amountOutMin_25Percent*/
            minAmounts[0], _deadline);

        IERC20(trustedPlatformTokenAddress).safeApprove(address(
            trustedStakingContractAddress), 0);
        IERC20(trustedPlatformTokenAddress).safeApprove(address(
            trustedStakingContractAddress), amountToDepositByContract);

        StakingContract(trustedStakingContractAddress).depositByContract(msg.sender,
            amountToDepositByContract, /*_amountOutMin_stakingReferralFee*/minAmounts
            [1], _deadline);

        uint half = _75Percent.div(2);
        uint otherHalf = _75Percent.sub(half);

        uint _rewardTokenReceived = doSwap(depositToken, trustedRewardTokenAddress, half
            , /*_amountOutMin_rewardTokenReceived*/minAmounts[4], _deadline);
        uint _baseTokenReceived = doSwap(depositToken, trustedBaseTokenAddress,
            otherHalf, /*_amountOutMin_baseTokenReceived*/minAmounts[5], _deadline);

        uint amountToDeposit = addLiquidityAndGetAmountToDeposit(
            _rewardTokenReceived,
            _baseTokenReceived,
            minAmounts,
            _deadline
```

```
        );

        require(amountToDeposit > 0, "Cannot deposit 0 Tokens");

        updateAccount(msg.sender, /*_amountOutMin_claimAsToken_dyp*/minAmounts[6], /*
            _amountOutMin_attemptSwap*/minAmounts[7], _deadline);

        depositedTokens[msg.sender] = depositedTokens[msg.sender].add(amountToDeposit);
        totalTokens = totalTokens.add(amountToDeposit);

        holders.add(msg.sender);
        depositTime[msg.sender] = now;
    }
```

Listing 3.3: `FarmProRata::deposit()`

This routine will swap half of the `75Percent-depositToken` into the `trustedRewardToken` and add the output tokens into the `trustedRewardToken/trustedBaseToken` pair to provide liquidity. However, the logic ignores the fact that the swap of the `depositToken` to the `trustedRewardToken` token will drive up the price of the `trustedRewardToken` token in `depositToken/trustedRewardToken` pair. In the case of `depositToken == trustedBaseToken`, this will lead to the result where certain `trustedBaseToken` tokens may be left in the contract after the calling of `addLiquidityAndGetAmountToDeposit()`.

Recommendation Refund the extra `tokenAmt - amountA` tokens to the `msg.sender` or do an optimization for the calculation of an optimal swap token amount.

Status The issue has been confirmed by the team. The team clarifies it is ok to remain a small amount of tokens on the contract because users will be able to make a vote through the governance and disburse the funds left from the swaps to the users that are still in the contracts based on their share.

3.3 Incompatibility with Deflationary Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ConstantReturnStaking`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `DYP v2` protocol, the `ConstantReturnStaking` contract is designed to take users' assets and deliver rewards depending on their deposit amounts. In particular, one interface, i.e., `stake()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `unstake()`, allows the user to withdraw the asset. For the above two operations, i.e., `stake()` and `unstake()`, the contract makes the use of `safeTransferFrom()` or `safeTransfer()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function stake(uint amountToStake, address referrer, uint _amountOutMin_referralFee,
         uint _amountOutMin_75Percent, uint _deadline) external noContractsAllowed
        notDuringEmergency {
        require(amountToStake > 0, "Cannot deposit 0 Tokens");
        IERC20(TRUSTED_DEPOSIT_TOKEN_ADDRESS).safeTransferFrom(msg.sender, address(this)
            , amountToStake);

        updateAccount(msg.sender, _amountOutMin_referralFee, _deadline);

        uint fee = amountToStake.mul(STAKING_FEE_RATE_X_100).div(1e4);
        uint amountAfterFee = amountToStake.sub(fee);
        if (fee > 0) {
            IERC20(TRUSTED_DEPOSIT_TOKEN_ADDRESS).safeTransfer(feeRecipientAddress, fee)
                ;
        }

        uint _75Percent = amountAfterFee.mul(75e2).div(100e2);

        uint contractDepositAmount = doSwap(TRUSTED_DEPOSIT_TOKEN_ADDRESS,
            TRUSTED_REWARD_TOKEN_ADDRESS, _75Percent, _amountOutMin_75Percent, _deadline
            );

        IERC20(TRUSTED_REWARD_TOKEN_ADDRESS).safeApprove(
            TRUSTED_BUYBACK_CONTRACT_ADDRESS, 0);
        IERC20(TRUSTED_REWARD_TOKEN_ADDRESS).safeApprove(
            TRUSTED_BUYBACK_CONTRACT_ADDRESS, contractDepositAmount);
        BuybackContract(TRUSTED_BUYBACK_CONTRACT_ADDRESS).depositByContract(msg.sender,
            contractDepositAmount);

        uint remainingAmount = amountAfterFee.sub(_75Percent);

        depositedTokens[msg.sender] = depositedTokens[msg.sender].add(remainingAmount);

        holders.add(msg.sender);

        if (referrals[msg.sender] == address(0)) {
            referrals[msg.sender] = referrer;
        }

        totalReferredAddressesOfUser[referrals[msg.sender]].add(msg.sender);
        activeReferredAddressesOfUser[referrals[msg.sender]].add(msg.sender);

        stakingTime[msg.sender] = now;
        emit Stake(msg.sender, remainingAmount);
    }
```

```
    function unstake(uint amountToWithdraw, uint _amountOutMin_referralFee, uint
        _deadline) external noContractsAllowed {
        require(depositedTokens[msg.sender] >= amountToWithdraw, "Invalid amount to
            withdraw");

        require(now.sub(stakingTime[msg.sender]) > LOCKUP_TIME, "You recently staked,
            please wait before withdrawing.");

        updateAccount(msg.sender, _amountOutMin_referralFee, _deadline);

        uint fee = amountToWithdraw.mul(UNSTAKING_FEE_RATE_X_100).div(1e4);
        uint amountAfterFee = amountToWithdraw.sub(fee);
        if (fee > 0) {
            IERC20(TRUSTED_DEPOSIT_TOKEN_ADDRESS).safeTransfer(feeRecipientAddress, fee)
                ;
        }
        IERC20(TRUSTED_DEPOSIT_TOKEN_ADDRESS).safeTransfer(msg.sender, amountAfterFee);

        depositedTokens[msg.sender] = depositedTokens[msg.sender].sub(amountToWithdraw);

        if (holders.contains(msg.sender) && depositedTokens[msg.sender] == 0) {
            holders.remove(msg.sender);
            activeReferredAddressesOfUser[referrals[msg.sender]].remove(msg.sender);
        }

        emit Unstake(msg.sender, amountToWithdraw);
    }
```

Listing 3.4: `ConstantReturnStaking::stake()/unstake()`

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

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into `DYP v2` staking contract for support.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom`

() call to ensure the book-keeping amount is accurate. An alternative solution is to regulate the set of ERC20 tokens that are permitted into the pool.

Status This issue has been confirmed. The team clarifies they won't add deflationary tokens into the pool as LP tokens.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `DYP-V2` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and tokens rescuing during emergency). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function claimAnyToken(address token, address recipient, uint amount) external
        onlyOwner {
        require(recipient != address(0), "Invalid Recipient");
        require(now > adminClaimableTime, "Contract not expired yet!");
        if (token == address(0)) {
            address payable _recipient = payable(recipient);
            _recipient.transfer(amount);
            return;
        }
        IERC20(token).safeTransfer(recipient, amount);
    }
```

Listing 3.5: `ConstantReturnStaking or ConstantReturnStaking_BuyBack::claimAnyToken()`

The above function allows the `owner` to withdraw all funds from the contract when `now > adminClaimableTime`. Note the `adminClaimableTime` is initialized from `now.add(adminCanClaimAfter)`, where `adminCanClaimAfter` could be a small period. If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the users.

Note a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, the `DYP v2` protocol involves the `Governance` contract and will let the `DAO`-like governance contract to take the privileged account. However, our analysis shows that not all privileged operations are going through the DAO-like governance process. In the following, we show example of privileged operations that are bypassing the governance process.

```
    function transferAnyERC20Token(address tokenAddress, address recipient, uint amount) external onlyOwner {
        require (tokenAddress != TRUSTED_TOKEN_ADDRESS  now > contractStartTime.add(ADMIN_CAN_CLAIM_AFTER), "Cannot Transfer Out main tokens!");
        require (Token(tokenAddress).transfer(recipient, amount), "Transfer failed!");
    }

    function transferAnyLegacyERC20Token(address tokenAddress, address recipient, uint amount) external onlyOwner {
        require (tokenAddress != TRUSTED_TOKEN_ADDRESS  now > contractStartTime.add(ADMIN_CAN_CLAIM_AFTER), "Cannot Transfer Out main tokens!");
        LegacyToken(tokenAddress).transfer(recipient, amount);
    }

    function transferAnyERC20TokenFromPool(address pool, address tokenAddress, address recipient, uint amount) external onlyOwner {
        StakingPool(pool).transferAnyERC20Token(tokenAddress, recipient, amount);
    }

    function transferAnyLegacyERC20TokenFromPool(address pool, address tokenAddress, address recipient, uint amount) external onlyOwner {
        StakingPool(pool).transferAnyOldERC20Token(tokenAddress, recipient, amount);
    }


    function declareEmergencyForContract(address trustedFarmContractAddress) external onlyOwner {
        StakingPool(trustedFarmContractAddress).declareEmergency();
    }
    function claimAnyTokenFromContract(address trustedFarmContractAddress, address token, address recipient, uint amount) external onlyOwner {
        StakingPool(trustedFarmContractAddress).claimAnyToken(token, recipient, amount);
    }
    function emergencyTransferContractOwnership(address trustedFarmContractAddress, address newOwner) external onlyOwner {
        require(isEmergency, "Can only execute this during emergency");
        StakingPool(trustedFarmContractAddress).transferOwnership(newOwner);
    }
```

Listing 3.6: `ConstantReturnStaking::multiple privileged operations related functions`

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 with the team. The team clarifies that the current governance was designed in a way to protect the users funds in case something bad happens, like an attack, this feature is useful in emergency situations. Also, the team clarifies that only one person has access to the cold wallet of the governance's admin. Meanwhile, when the contract is stable, they will transfer the owner of `Governance` to a multi-sig or `timelock` contract.

3.5 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Governance`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

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

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below.

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

Listing 3.7: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `Token` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(`

address, uint)external returns (bool). As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted.

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

In the following, we show the transferAnyERC20Token() routine in the Governance contract. If USDT is given as token, the unsafe version of Token(tokenAddress).transfer(recipient, amount) (line 618) may revert as there is no return value in the USDT token contract's transfer() implementation (but the Token interface expects a return value)!

```
616     function transferAnyERC20Token(address tokenAddress, address recipient, uint amount)
             external onlyOwner {
617         require (tokenAddress != TRUSTED_TOKEN_ADDRESS  now > contractStartTime.add(
                ADMIN_CAN_CLAIM_AFTER), "Cannot Transfer Out main tokens!");
618         require (Token(tokenAddress).transfer(recipient, amount), "Transfer failed!");
619     }
```

Listing 3.8: Governance::transferAnyERC20Token()

Note that other routines removeVotes(), withdrawAllTokens() and addVotes() share the same issue.

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

Status This issue has been confirmed. The team clarifies the protocol will not support non-compliant tokens.

4 Conclusion

In this audit, we have analyzed the design and implementation of `DYP-v2`, which contains several separate products including staking, buyback, farm and governance. The current code base is well organized and 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-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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

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

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