SMART CONTRACT AUDIT REPORT

for

DEFI YIELD PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China

Dec. 15, 2020

Document Properties

Client	DeFi Yield Protocol
Title	Smart Contract Audit Report
Target	DeFi Yield Protocol
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Chiachih Wu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Dec. 15, 2020	Xudong Shao	Final Release

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction **5**

1.1 About DeFi Yield Protocol 5

1.2 About PeckShield 6

1.3 Methodology 6

1.4 Disclaimer 8

2 Findings **10**

2.1 Summary 10

2.2 Key Findings 11

3 Detailed Results **12**

3.1 Missed Sanity Check in Governance::noContractsAllowed() 12

3.2 Inconsistent DYP Disbursed in Staking::disburseTokens() 13

3.3 Unfair Token Lockup Mechanism 15

3.4 Unsafe Ownership Transition 17

3.5 Unused Functions in Interfaces 18

4 Conclusion **22**

5 Appendix **23**

5.1 Basic Coding Bugs 23

5.1.1 Constructor Mismatch 23

5.1.2 Ownership Takeover 23

5.1.3 Redundant Fallback Function 23

5.1.4 Overflows & Underflows 23

5.1.5 Reentrancy 24

5.1.6 Money-Giving Bug 24

5.1.7 Blackhole 24

5.1.8 Unauthorized Self-Destruct 24

5.1.9 Revert DoS 24

5.1.10 Unchecked External `Call` 25
5.1.11 Gasless `Send` 25
5.1.12 `Send` Instead Of `Transfer` 25
5.1.13 Costly Loop 25
5.1.14 (Unsafe) Use Of Untrusted Libraries 25
5.1.15 (Unsafe) Use Of Predictable Variables 26
5.1.16 Transaction Ordering Dependence 26
5.1.17 Deprecated Uses 26
5.2 Semantic Consistency Checks 26
5.3 Additional Recommendations 26
5.3.1 Avoid Use of Variadic Byte Array 26
5.3.2 Make Visibility Level Explicit 27
5.3.3 Make Type Inference Explicit 27
5.3.4 Adhere To Function Declaration Strictly 27

References **28**

1 Introduction

Given the opportunity to review the **DeFi Yield Protocol** design document and related smart contract source code, 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. This document outlines our audit results.

1.1 About DeFi Yield Protocol

DeFi Yield Protocol (DYP) is a unique platform that allows anyone to provide liquidity and to be rewarded on Ethereum. And 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 at 00:00 UTC, and the ETH is distributed as a reward to the liquidity providers.

The basic information of DeFi Yield Protocol is as follows:

Table 1.1: Basic Information of DeFi Yield Protocol

Item	Description
Issuer	DeFi Yield Protocol
Website	https://dyp.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Dec. 15, 2020

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

- https://github.com/dypfinance/DYP-staking-governance-dapp (46fe7bd)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 [7]:

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

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

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) [6], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the DeFi Yield Protocol Protocol design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	0	
Low	1	■
Informational	4	■ ■ ■ ■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 low-severity vulnerability and 4 informational recommendations.

Table 2.1: Key DeFi Yield Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Missed Sanity Check in Governance::noContractsAllowed()	Coding Practices	Confirmed
PVE-002	Info.	Inconsistent DYP Disbursed in Staking::disburseTokens()	Business Logic	Confirmed
PVE-003	Info.	Unfair Token Lockup Mechanism	Business Logic	Confirmed
PVE-004	Info.	Unsafe Ownership Transition	Business Logic	Confirmed
PVE-005	Info.	Unused Functions in Interfaces	Coding Practices	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 Missed Sanity Check in Governance::noContractsAllowed()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Governance`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

Description

In DeFi Yield Protocol, the `Staking` contract allows users to stake `Uniswap LP Tokens` to receive WETH and DYP tokens as rewards. And the `Governance` contract manages proposals for staking pools. In order to reduce the risks of the contract being attacked by malicious users, the key functions in these two contracts use `noContractsAllowed` modifier to prevent contracts calling them directly.

```
    modifier noContractsAllowed() {
        require(!(address(msg.sender).isContract()), "No Contracts Allowed!");
        _;
    }
```

Listing 3.1: governance.sol

The internal function, `isContract`, called by `noContractsAllowed` determines whether the caller is a contract by checking the `extcodesize` of the caller (line 62). However, a contract does not have source code available during construction. So, if a contract makes calls to other contracts inside the `constructor()`, the `extcodesize` would be 0, which makes the caller passes the `noContractAllowed` check.

```
55     function isContract(address account) internal view returns (bool) {
56         // This method relies on extcodesize, which returns 0 for contracts in
57         // construction, since the code is only stored at the end of the
58         // constructor execution.
59 
60         uint256 size;
61         // solhint-disable-next-line no-inline-assembly
```

```
        assembly { size := extcodesize(account) }
        return size > 0;
    }
```

Listing 3.2: governance.sol

Recommendation Ensure the `msg.sender` is the same as `tx.origin`.

```
    modifier noContractsAllowed() {
        require(!(address(msg.sender).isContract()) && tx.origin == msg.sender, "No
            Contracts Allowed!");
        _;
    }
```

Listing 3.3: governance.sol

Status This issue has been fixed in the commit: 92c497f0ff831e55b0b93a57d82b65604526ede1.

3.2 Inconsistent DYP Disbursed in Staking::disburseTokens()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Staking`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As we introduced in Section 3.1, DeFi Yield Protocol has two modules: `Staking` and `Governance`. Users can stake and withdraw `Uniswap LP Tokens` on `Staking` contract for farming WETH and DYP rewards. In the current implementation, the amount of DYP to be paid out is calculated according to the time passed since the contract is deployed. Specifically, the `disburseTokens()` function gets the pending disbursement `amount` through `getPendingDisbursement()` (line 1004). Later on, `tokensToBeSwapped` and `contractBalance` are updated (line 1011 and line 1013) followed by setting `lastDisburseTime` to the current timestamp in line 1014.

```
1003    function disburseTokens() private {
1004        uint amount = getPendingDisbursement();
1005
1006        if (contractBalance < amount) {
1007            amount = contractBalance;
1008        }
1009        if (amount == 0  totalTokens == 0) return;
1010
```

```
        tokensToBeSwapped = tokensToBeSwapped.add(amount);

        contractBalance = contractBalance.sub(amount);
        lastDisburseTime = now;
    }
```

Listing 3.4: Staking.sol

Inside `getPendingDisbursement()`, the `timeDiff` is derived by the current timestamp and the `lastDisburseTime`. The `timeDiff` is later used to calculate the `pendingDisburse` which is returned to `disburseTokens()`.

```
    function getPendingDisbursement() public view returns (uint) {
        uint timeDiff;
        uint _now = now;
        uint _stakingEndTime = contractDeployTime.add(disburseDuration);
        if (_now > _stakingEndTime) {
            _now = _stakingEndTime;
        }
        if (lastDisburseTime >= _now) {
            timeDiff = 0;
        } else {
            timeDiff = _now.sub(lastDisburseTime);
        }

        uint pendingDisburse = disburseAmount
                                    .mul(disbursePercentX100)
                                    .mul(timeDiff)
                                    .div(disburseDuration)
                                    .div(10000);
        return pendingDisburse;
    }
```

Listing 3.5: Staking.sol

While reviewing the implementation, we find out couple corner cases that make the distribution inconsistent. In particular, when `contractBalance` is 0, the amount of DYP to be disbursed would be 0 without setting the `lastDisburseTime`. This makes the period of time be counted into `timeDiff` again when the contract admin adds contract balance by calling `addContractBalance()`. However, if the contract balance is larger than 0 but less than the `amount` in `disburseTokens()`, the amount of DYP to be disbursed would be the contract balance, which is less than the amount that is supposed to be disbursed according to the calculation. Therefore, the tokens to be disbursed varies when the contract balance is changed.

Recommendation Update the `lastDisburseTime` when contract balance is 0.

Status This issue has been confirmed by the team. However, the contract balance is supposed to be added only once. Since it's not a security issue, the dev team decides to leave it as is.

3.3 Unfair Token Lockup Mechanism

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Governance`,`Staking`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Governance` contract allows users to submit proposals for pools and add/remove votes for proposals. An open proposal is allowed to be voted for `VOTE_DURATION`. A voter cannot get their tokens back until the voted proposal is closed for voting. Besides, only after the latest proposal that each user votes for is closed, can that user withdraws her tokens. Specifically, the `withdrawAllTokens()` function checks if the current timestamp (i.e., `now`) is greater than the start time of the last voted proposal + the `VOTE_DURATION` (i.e., 5 minutes) (line 419) before transferring `TRUSTED_TOKEN_ADDRESS` to the caller.

```
418     function withdrawAllTokens() external noContractsAllowed {
419         require(now > lastVotedProposalStartTime[msg.sender].add(VOTE_DURATION), "Tokens
                 are still in voting!");
420         require(Token(TRUSTED_TOKEN_ADDRESS).transfer(msg.sender, totalDepositedTokens[
                msg.sender]), "transfer failed!");
421         totalDepositedTokens[msg.sender] = 0;
422     }
```

Listing 3.6: governance.sol

However, `lastVotedProposalStartTime[msg.sender]` is updated every time the user `addVotes()` for a particular open proposal (line 392). If an user votes more than once, the `lastVotedProposalStartTime[msg.sender]` depends on her voted proposal with the latest start time. This is unfair because the tokens locked for old proposals could be locked for more than `VOTE_DURATION`.

```
368     function addVotes(uint proposalId, Option option, uint amount) external
            noContractsAllowed {
369         require(amount > 0, "Cannot add 0 votes!");
370         require(isProposalOpen(proposalId), "Proposal is closed!");
371
372         require(Token(TRUSTED_TOKEN_ADDRESS).transferFrom(msg.sender, address(this),
                amount), "transferFrom failed!");
373
374         // if user is voting for this proposal first time
375         if (votesForProposalByAddress[msg.sender][proposalId] == 0) {
376             votedForOption[msg.sender][proposalId] = option;
377         } else {
378             if (votedForOption[msg.sender][proposalId] != option) {
379                 revert("Cannot vote for both options!");
```

```
            }
        }

        if (option == Option.ONE) {
            optionOneVotes[proposalId] = optionOneVotes[proposalId].add(amount);
        } else {
            optionTwoVotes[proposalId] = optionTwoVotes[proposalId].add(amount);
        }
        totalDepositedTokens[msg.sender] = totalDepositedTokens[msg.sender].add(amount);
        votesForProposalByAddress[msg.sender][proposalId] = votesForProposalByAddress[
            msg.sender][proposalId].add(amount);

        if (lastVotedProposalStartTime[msg.sender] < proposalStartTime[proposalId]) {
            lastVotedProposalStartTime[msg.sender] = proposalStartTime[proposalId];
        }
    }
```

Listing 3.7: governance.sol

Similar logic applies to the `deposit()` and `withdraw()` function in the `Staking` contract. Every time the user deposits, her `depositTime[msg.sender]` would be updated (line 937).

```
924     function deposit(uint amountToDeposit) public noContractsAllowed {
925         require(amountToDeposit > 0, "Cannot deposit 0 Tokens");
926 
927         updateAccount(msg.sender);
928 
929         require(Token(trustedDepositTokenAddress).transferFrom(msg.sender, address(this)
            , amountToDeposit), "Insufficient Token Allowance");
930 
931         depositedTokens[msg.sender] = depositedTokens[msg.sender].add(amountToDeposit);
932         totalTokens = totalTokens.add(amountToDeposit);
933 
934         if (!holders.contains(msg.sender)) {
935             holders.add(msg.sender);
936         }
937         depositTime[msg.sender] = now;
938     }
```

Listing 3.8: Staking.sol

As implemented in the `withdraw()` function, the user cannot withdraw her tokens back until `cliffTime` after the last batch of `deposit()` (line 945). In the case that the user `deposit()` more than once, an older batch of deposited tokens could be locked for more than `cliffTime`, which is again unfair.

```
941     function withdraw(uint amountToWithdraw) public noContractsAllowed {
942         require(amountToWithdraw > 0, "Cannot withdraw 0 Tokens!");
943 
944         require(depositedTokens[msg.sender] >= amountToWithdraw, "Invalid amount to
            withdraw");
945         require(now.sub(depositTime[msg.sender]) > cliffTime, "You recently deposited,
            please wait before withdrawing.");
```

```
        updateAccount(msg.sender);

        require(Token(trustedDepositTokenAddress).transfer(msg.sender, amountToWithdraw), "Could not transfer tokens.");

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

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

Listing 3.9: Staking.sol

Recommendation Allocate a lock ID for each user whenever she votes/deposits and judge whether she can withdraw by the ID.

Status This issue has been confirmed by the team. However, since current implementation is the simplest and most straightforward lockup solution, the dev team decides to leave it as is.

3.4 Unsafe Ownership Transition

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Owned`, `Staking`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In DeFi Yield Protocol, the `Owned` contract is used for ownership management in `Staking` contract. When the contract owner needs to transfer the ownership to another address, she could invoke the `transferOwnership()` function with a `newOwner` address.

```
function transferOwnership(address newOwner) onlyOwner public {
    require(newOwner != address(0));
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
}
```

Listing 3.10: Staking.sol

However, if the `newOwner` is not the exact address of the new owner (e.g., due to a typo), nobody could own that contract anymore.

Recommendation Implement a two-step ownership transfer mechanism that allows the new owner to claim the ownership by signing a transaction. An example is shown below:

```
function transferOwnership(
    address newOwner
)
    external
    onlyOwner
{
    require(newOwner != address(0), "Owned: Address must not be null");
    require(candidateOwner != newOwner, "Owned: Same candidate owner");
    candidateOwner = newOwner;
}

function claimOwner()
    external
{
    require(candidateOwner == msg.sender, "Owned: Claim ownership failed");
    owner = candidateOwner;
    emit OwnerChanged(candidateOwner);
}
```

Listing 3.11: Staking.sol

Status This issue has been confirmed by the team. However, this function will be used manually only once while setting up the staking to transfer ownership to the governance, after that proposals are created on the governance side and everyone gets ample time to review the new owner address and vote for the change of ownership of the staking contract. So the dev team decides to leave it as is.

3.5 Unused Functions in Interfaces

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Staking`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

Description

In the `Staking` contract, there are some unused functions in interfaces such as IUniswapV2Router01, IUniswapV2Router02, etc.

`addLiquidity()`, `addLiquidityETH()`, `removeLiquidity()`, `removeLiquidityETH()`, etc.

```
interface IUniswapV2Router01 {
    function factory() external pure returns (address);
```

```
    function WETH() external pure returns (address);

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB, uint liquidity);
    function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external payable returns (uint amountToken, uint amountETH, uint liquidity);
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB);
    function removeLiquidityETH(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external returns (uint amountToken, uint amountETH);
    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountA, uint amountB);
    function removeLiquidityETHWithPermit(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
```

```
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountToken, uint amountETH);
    function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);
    function swapTokensForExactTokens(
        uint amountOut,
        uint amountInMax,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);
    function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address
        to, uint deadline)
        external
        payable
        returns (uint[] memory amounts);
    function swapTokensForExactETH(uint amountOut, uint amountInMax, address[] calldata
        path, address to, uint deadline)
        external
        returns (uint[] memory amounts);
    function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata
        path, address to, uint deadline)
        external
        returns (uint[] memory amounts);
    function swapETHForExactTokens(uint amountOut, address[] calldata path, address to,
        uint deadline)
        external
        payable
        returns (uint[] memory amounts);

    function quote(uint amountA, uint reserveA, uint reserveB) external pure returns (
        uint amountB);
    function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) external pure
        returns (uint amountOut);
    function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) external pure
        returns (uint amountIn);
    function getAmountsOut(uint amountIn, address[] calldata path) external view returns
         (uint[] memory amounts);
    function getAmountsIn(uint amountOut, address[] calldata path) external view returns
         (uint[] memory amounts);
}
```

Listing 3.12: Staking.sol

Recommendation Remove the unused functions in interfaces.

Status This issue has been confirmed by the team. However, removal of code will be saving gas only during the contract deployment. And gas savings during contract deployment are pretty less than the development cost involved in removing the code and testing again. So the dev team decides to leave it as is.

4 Conclusion

In this audit, we thoroughly analyzed the DeFi Yield Protocol (DYP) design and implementation. DYP is a unique protocol that allows virtually any user to provide liquidity, earn DYP tokens as yield while maintaining the token price. During the audit, we notice that the current code base is well structured and neatly organized, and those identified issues are promptly confirmed and fixed.

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

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [8, 9, 10, 11, 13].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [14] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

[8] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[9] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[10] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[11] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[13] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[14] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.