SMART CONTRACT AUDIT REPORT

for

DYP EARN VAULT

PeckShield

Prepared By: Shuxiao Wang

PeckShield

April 22, 2021

Document Properties

Client	DeFi Yield Protocol
Title	Smart Contract Audit Report
Target	DYP Earn Vault
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 22, 2021	Xuxian Jiang	Final Release
1.0-rc	April 13, 2021	Xuxian Jiang	Release Candidate
0.2	April 9, 2021	Xuxian Jiang	Additional Findings
0.1	April 7, 2021	Xuxian Jiang	Initial Draft

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

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **DYP Earn Vault** protocol, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About DYP Earn Vault

The `DYP Earn Vault` is an automated yield farming contract that allows users to deposit a particular token, for which the protocol automates yield farming strategies by moving providers funds between the most profitable platforms. Of the profits, 75% is converted to ETH and distributed to the liquidity providers, while the remaining 25% is used to buy back the protocol governance token in order to add liquidity and maintain the token price stability.

The basic information of DYP Earn Vault is as follows:

Table 1.1: Basic Information of DYP Earn Vault

Item	Description
Issuer	DeFi Yield Protocol
Website	https://dyp.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 22, 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/dyp-earn-vault.git (aa3bce5)

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

- https://github.com/dypfinance/dyp-earn-vault.git (bc7435d)

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	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

PeckShield

1.3 Methodology

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

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

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

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

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

PeckShield

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.

PeckShield

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 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 DYP Earn Vault 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	2	🟦 🟦
Low	1	🟦
Informational	0	
Total	3	

PeckShield

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 2 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key DYP Earn Vault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Nonfunctional Slippage Control	Coding Practices	Fixed
PVE-002	Medium	Possible Sandwich/MEV For Maximum PlatformTokenDivs	Time and State	Confirmed
PVE-003	Low	Simplified noContractsAllowed() Implementation	Business Logic	Fixed

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.

PeckShield

3 Detailed Results

3.1 Nonfunctional Slippage Control

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Vault, VaultWETH
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The DYP Earn Vault has a built-in integration with Compound. With the integration, users-deposited funds are forwarded to Compound for respective cTokens held by the DYP Earn Vault. Users may withdraw their cTokens back to their deposit and interest from Compound. Note the withdrawal operation has a 0.3% withdrawal fee in respective token: 25% of withdrawal fee is used to buy back the protocol token from Uniswap and 75% distributed pro-rata among active vault users. The protocol setup assumes Compound and Uniswap have the appropriate amount of liquidity available for the whole setup to work properly.

In the following, we examine the Vault::handleFee() routine that is designed to process the withdrawal fee. After proper calculation of withdrawal fee, this routine converts the fee from the deposited token to the platform token via swapExactTokensForTokens in Uniswap (line 1269). It comes to our attention that the routine attempts to limit possible slippage by computing minimum amount after conversion in amountOutMin (line 1266).

```
1247    function handleFee(uint feeAmount) private {
1248        uint buyBackFeeAmount = feeAmount.mul(FEE_PERCENT_TO_BUYBACK_X_100).div(
                ONE_HUNDRED_X_100);
1249        uint remainingFeeAmount = feeAmount.sub(buyBackFeeAmount);

1251        // handle distribution
1252        distributeTokenDivs(remainingFeeAmount);
```

```
        // handle buyback
        // --- swap token to platform token here! ----
        IERC20(TRUSTED_DEPOSIT_TOKEN_ADDRESS).safeApprove(address(uniswapRouterV2), 0);
        IERC20(TRUSTED_DEPOSIT_TOKEN_ADDRESS).safeApprove(address(uniswapRouterV2),
            feeAmount);

        uint oldPlatformTokenBalance = IERC20(TRUSTED_PLATFORM_TOKEN_ADDRESS).balanceOf(
            address(this));
        address[] memory path = new address[](3);
        path[0] = TRUSTED_DEPOSIT_TOKEN_ADDRESS;
        path[1] = uniswapRouterV2.WETH();
        path[2] = TRUSTED_PLATFORM_TOKEN_ADDRESS;

        uint estimatedAmountOut = uniswapRouterV2.getAmountsOut(buyBackFeeAmount, path)
            [2];
        uint amountOutMin = estimatedAmountOut.mul(ONE_HUNDRED_X_100.sub(
            SLIPPAGE_TOLERANCE_X_100)).div(ONE_HUNDRED_X_100);

        uniswapRouterV2.swapExactTokensForTokens(buyBackFeeAmount , amountOutMin, path,
             address(this), block.timestamp);
        uint newPlatformTokenBalance = IERC20(TRUSTED_PLATFORM_TOKEN_ADDRESS).balanceOf(
            address(this));
        uint platformTokensReceived = newPlatformTokenBalance.sub(
            oldPlatformTokenBalance);
        IERC20(TRUSTED_PLATFORM_TOKEN_ADDRESS).safeTransfer(BURN_ADDRESS,
            platformTokensReceived);
        // ---- end swap token to plaform tokens -----
    }
```

Listing 3.1: Vault::handleFee()

PeckShield

We point out that the computation of amountOutMin is performed via getAmountsOut() in the same Uniswap, which means the minimum amount after conversion is always satisfied. In other words, current slippage control is not functional. A proper slippage control may require manual input or a separate pricing oracle to compare and restrict potential deviation from Uniswap. Note that both Vault and VaultWETH contracts share the same issue.

Recommendation Properly apply slippage control to avoid unnecessary conversion loss from potential price manipulation in Uniswap.

Status This issue has been fixed in this commit: bc7435de.

3.2 Possible Sandwich/MEV For Maximum PlatformTokenDivs

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Vault`, `VaultWETH`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

As mentioned in Section 3.1, the `DYP Earn Vault` has a built-in integration with `Compound` to earn interests. Also, when a user withdraws from the protocol, there is an associated withdrawal fee, which is partially used to buy back the protocol token in order to add liquidity and maintain the token price stability. In the following, we examine the platform token dividends that may be claimed by participating users.

To elaborate, we show below the full implementation of the `_claimPlatformTokenDivs()` routine. This routine firstly retrieves the current dividend balance of the requesting user (in the deposited tokens - line 1125), then queries `Uniswap` for possible converted amount in the platform tokens (line 1135), and finally transfers out the queried amount (line 1137).

```
1124     function _claimPlatformTokenDivs() private {
1125         updateAccount(msg.sender);
1126         uint amount = platformTokenDivsBalance[msg.sender];
1127         platformTokenDivsBalance[msg.sender] = 0;
1128         if (amount == 0) return;
1129 
1130         address[] memory path = new address[](3);
1131         path[0] = TRUSTED_DEPOSIT_TOKEN_ADDRESS;
1132         path[1] = uniswapRouterV2.WETH();
1133         path[2] = TRUSTED_PLATFORM_TOKEN_ADDRESS;
1134 
1135         uint estimatedAmountOut = uniswapRouterV2.getAmountsOut(amount, path)[2];
1136         decreaseTokenBalance(TRUSTED_PLATFORM_TOKEN_ADDRESS, estimatedAmountOut);
1137         IERC20(TRUSTED_PLATFORM_TOKEN_ADDRESS).safeTransfer(msg.sender,
             estimatedAmountOut);
1138         totalEarnedPlatformTokenDivs[msg.sender] = totalEarnedPlatformTokenDivs[msg.
             sender].add(estimatedAmountOut);
1139 
1140         emit PlatformTokenRewardClaimed(msg.sender, estimatedAmountOut);
1141     }
```

Listing 3.2: Vault::_claimPlatformTokenDivs()

We notice the collected dividends are (virtually) routed to `UniswapV2` in order to swap them to the platform token as dividends. And the `getAmountsOut()` calculation does not have any restriction on possible slippage and is therefore vulnerable to possible sandwich attacks, resulting in a smaller gain for this round of yielding. Note that both Vault and VaultWETH contracts share the same issue.

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

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

Status This issue has been confirmed. The current implementation has the noContractsAllowed modifier in place to block contract-based calls. We need to admit that it mitigates potential sandwich-based attacks, but does not eliminate the issue.

3.3 Simplified noContractsAllowed() Implementation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Vault, VaultWETH
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

PeckShield

Description

As mentioned in Section 3.2, the protocol has designed a modifier noContractsAllowed() to limit a number of claim-related calls to EOA-based accounts only.

To elaborate, we show below its noContractsAllowed() function. In essence, it ensures the caller is not a contract and should equal to the transaction initiator. Note that the equality between the transaction initiator and the caller guarantees that the caller is not a contract, which makes it redundant to check that the caller is not a contract.

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

Listing 3.3: Vault :: noContractsAllowed()

Note that both `Vault` and `VaultWETH` contracts share the same issue.

Recommendation Simplify the `noContractsAllowed()` logic in a succinct form to remove redundant computation.

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

Listing 3.4: Vault :: noContractsAllowed()

Status This issue has been fixed in this commit: `bc7435de`.

PeckShield

4 Conclusion

In this audit, we have analyzed the DYP Earn Vault design and implementation. The system presents a unique offering as an automated yield farming contract that allows users to deposit a particular token, for which the protocol automates yield farming strategies by moving providers funds between the most profitable platforms. 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.

PeckShield

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.

PeckShield