SMART CONTRACT AUDIT REPORT

for

Olive(formerly Polysynth)

Prepared By: Xiaomi Huang

PeckShield

December 6, 2022

Document Properties

Client	Olive
Title	Smart Contract Audit Report
Target	Olive
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 6, 2022	Jing Wang	Final Release
1.0-rc	November 28, 2022	Jing Wang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the Olive 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 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 Olive

Olive is a decentralized protocol offering yield generating principal protection notes, structured products and option vaults on EVM chains such as Ethereum, Arbitrum and Polygon. It is different from other option vault protocols as it offers principal protected notes, which offers principal protected yield with zero credit risk. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Olive

Item	Description
Issuer	Olive
Website	https://oliveapp.finance/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 6, 2022

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

- https://github.com/kryptolabs/dov-audit-contracts (fd2e12e)

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

- https://github.com/kryptolabs/dov-audit-contracts (ca98606)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

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

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

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

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered 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 implementation of the `Olive` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	0	
Informational	1	■
Total	3	

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

2.2 Key Findings

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

Table 2.1: Key Olive Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Fee Handling Of Deposit	Business Logic	Fixed
PVE-002	Informational	Inconsistency in Decimal Handling Between Functions	Business Logic	Fixed
PVE-003	High	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improper Fee Handling Of Deposit

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

Description

The PolysynthVault contract provides two routines for user to deposit assets into the vault. While reviewing the implementation, we notice that the depositFee is not properly handled, which could be bypassed in certain cases. To illustrate, we show below the related routines.

```
    function depositETH() external payable nonReentrant {
        require(vaultParams.asset == WETH, "!WETH");
        require(msg.value > 0, "!value");

        _depositFor(msg.value, msg.sender);

        IWETH(WETH).deposit{value: msg.value}();
    }

    function deposit(uint256 amount) external nonReentrant {
        require(amount > 0, "!amount");

        _depositFor(amount, msg.sender);

        if (depositFee>0){
            uint256 fee = amount.mul(depositFee).div(100 * Vault.FEE_MULTIPLIER);
            amount += fee;
        }

        // An approve() by the msg.sender is required beforehand
        IERC20(vaultParams.asset).safeTransferFrom(
            msg.sender,
```

```
            address(this),
            amount
        );
    }
```

Listing 3.1: `OliveVault::depositETH()and deposit()`

It comes to our attention that the `depositFee` is charged in the `deposit()` routine, but not in the `depositETH()` routine. This will cause inconsistency in the fee charging and a normal user could bypass the fee charging by calling the `depositETH()` routine.

Recommendation Revisit the above logic of `depositETH()` to handle the `depositFee` properly.

Status The issue has been fixed by this commit: `ca98606`.

3.2 Inconsistency in Decimal Handling Between Functions

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Swap`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In `Ethereum`, tokens are designed to have fixed decimals, e.g., the `OToken` token contract has a decimal of 8. During our analysis with the `Swap` contract, we notice the contract defines a constant variable `OTOKEN_DECIMALS` to help the calculations related with `OToken` decimal when needed. To elaborate, we show below the related routines in the `Swap` contract.

```
function averagePriceForOffer(uint256 swapId)
  external
  view
  override
  returns (uint256)
{
  Offer storage offer = swapOffers[swapId];
  require(offer.totalSize != 0, "Offer does not exist");

  uint256 availableSize = offer.availableSize;

  // Deduct the initial 1 wei offset if offer is not fully settled
  uint256 adjustment = availableSize != 0 ? 1 : 0;

  return
      ((offer.totalSales - adjustment) * (10**8)) /
      (offer.totalSize - availableSize);
```

```
  }

  function _swap(
    OfferDetails memory details,
    Offer storage offer,
    Bid calldata bid
) internal {
    ...
    require(_markNonceAsUsed(signatory, bid.nonce), "NONCE_ALREADY_USED");
    require(
        bid.buyAmount <= offer.availableSize,
        "BID_EXCEED_AVAILABLE_SIZE"
    );
    require(bid.buyAmount >= details.minBidSize, "BID_TOO_SMALL");

    // Ensure min. price is met
    uint256 bidPrice =
        (bid.sellAmount * 10**OTOKEN_DECIMALS) / bid.buyAmount;
    require(bidPrice >= details.minPrice, "PRICE_TOO_LOW");
    ...
}
```

Listing 3.2: `Swap::averagePriceForOffer()`

It comes to our attention that the `averagePriceForOffer()` function uses the hard-coded value `(10**8)` instead of the constant value `OTOKEN_DECIMALS`, while the `_swap()` function uses the constant value `OTOKEN_DECIMALS` in the calculation of price. We suggest keeping the decimal handling consistent in the two routines mentioned above.

Recommendation Ensure the consistency of decimal handling between different functions.

Status The issue has been fixed by this commit: `ca98606`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple files`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Olive` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration). 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 `owner` account and its related privileged accesses in current contract.

To elaborate, we show the `setBorrower()` from the `KikoVault` contract. This function allows the `owner` account to set the `borrower` who can take all funds from the vault without collateral.

```
  function setBorrower(address _borrower) external onlyOwner {
    require(_borrower != address(0), "!_borrower");
    borrower = _borrower;
  }

  function borrow() external nonReentrant {
    require(!optionState.isBorrowed,"already borrowed");
    require(msg.sender == borrower, "unauthorised");

    uint256 borrowAmount = uint256(vaultState.lockedAmount).mul(optionState.borrowRate).
        div(Kiko.RATIO_MULTIPLIER);
    if (borrowAmount > 0) {
        transferAsset(msg.sender, borrowAmount);
    }

    // Event for borrow amount
    emit Borrow(borrower, borrowAmount, optionState.borrowRate);

    optionState.isBorrowed = true;
 }
```

Listing 3.3: KikoVault::setBorrower()

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 mitigated. The team clarifies they have `OTC` operations for the vaults auction and the `Owner` will whitelist the address to borrow funds based on the highest bidder. Also, making owner as `DAO` with timelocks account might slow down the fund release process, which is undesirable as market maker need to withdraw funds as soon as they win the auction (typically within mins) to hedge their positions before meaningful price change. The teams confirm the `Owner` account will start with `EOA` and will be transferred to multi-sig when protocol works as expected.

4 Conclusion

In this audit, we have analyzed the `Olive` design and implementation. `Olive` is a decentralized protocol offering yield generating structured products and option vaults on `Ethereum` and `Polygon` blockchains. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

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

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

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

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

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

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

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

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