SMART CONTRACT AUDIT REPORT

for

Polynomial Earn (v2)

Prepared By: Patrick Lou

PeckShield

July 31, 2022

Document Properties

Client	Polynomial
Title	Smart Contract Audit Report
Target	Polynomial Earn
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	July 31, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Polynomial Earn (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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Polynomial Earn

The Polynomial Earn is designed to receive asset from depositors and invest its full asset in a so-called weekly options strategy. In essence, it sells the newly minted options to Lyra AMM in batches to collect possible yields. If the option that is sold in the strategy expired out of the money, the premium is collected and distributed to the depositors. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Polynomial Earn

Item	Description
Name	Polynomial
Website	https://www.polynomial.fi/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 31, 2022

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

- https://github.com/Polynomial-Protocol/earn-contracts-v2.git (4d1b715)

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

- https://github.com/Polynomial-Protocol/earn-contracts-v2.git (664e61b)

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 the 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.

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

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

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. 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 `Polynomial Earn (v2)` smart 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 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	2	■ ■
Informational	0	
Total	4	

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 2 low-severity vulnerabilities.

Table 2.1: Key Polynomial Earn Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Option Settlement in Put/-CallSellingVault	Business Logic	Resolved
PVE-002	Low	Improved Precision in processWithdrawalQueue()	Numeric Errors	Resolved
PVE-003	Low	Revisited StrikeId Removal in Put-SellingVault	Coding Practices	Resolved
PVE-004	Medium	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 Option Settlement in Put/CallSellingVault

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Put/CallSellingVault`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Polynomial Earn` protocol has developed a number of vaults, which are used to open, close, or settle options. While analyzing two these vaults, i,e., `PutSellingVault` and `CallSellingVault`, we notice their option settlement logic can be improved.

To elaborate, we show below the implementation of the `_settleOptions()` function from the `PutSellingVault` contract. As the name indicates, this function iterates the given set of `_strikeIds` for settlement. During the settlement, various accounting information is accordingly updated. Specifically, when the `option premium` is collected, there are two cases: (1) `positionData.premiumCollected > 0` (lines 722-728) and (2) `positionData.premiumCollected <=0` (line 729). It comes to our attention that the second case is not handled properly. In particular, it is currently updated as `totalFunds -= uint256(positionData.premiumCollected)` (line 729), which should be revised as `totalFunds -= uint256 (-positionData.premiumCollected)`.

```
692     function _settleOptions(uint256[] memory _strikeIds) internal {
693         for (uint256 i = 0; i < _strikeIds.length; i++) {
694             PositionData storage positionData = positionDatas[_strikeIds[i]];
695 
696             if (positionData.amount == 0) {
697                 revert ExpectedNonZero();
698             }
699 
700             OptionToken.PositionState optionState = OPTION_TOKEN.getPositionState(
                    positionData.positionId);
```

```
            if (optionState != OptionToken.PositionState.SETTLED) {
                revert OptionNotSettled(_strikeIds[i], positionData.positionId,
                    optionState);
            }

            (
                uint256 strikePrice,
                uint256 priceAtExpiry,
            ) = MARKET.getSettlementParameters(_strikeIds[i]);

            if (priceAtExpiry == 0) {
                revert InvalidExpiryPrice();
            }

            uint256 ammProfit = (priceAtExpiry < strikePrice) ? (strikePrice -
                priceAtExpiry).mulWadDown(positionData.amount) : 0;

            if (ammProfit > 0) {
                totalFunds -= ammProfit;
            }

            usedFunds -= positionData.collateral;

            if (positionData.premiumCollected > 0) {
                uint256 profit = uint256(positionData.premiumCollected);
                uint256 perfFees = profit.mulWadDown(performanceFee);
                ERC20(SUSD).safeTransfer(feeReceipient, perfFees);
                totalFunds += (profit - perfFees);
                totalPremiumCollected -= profit;
            } else {
                totalFunds -= uint256(positionData.premiumCollected);
            }

            emit SettleOption(
                _strikeIds[i],
                positionData.positionId,
                positionData.amount,
                positionData.collateral,
                positionData.premiumCollected,
                ammProfit
            );

            positionData.premiumCollected = 0;
            positionData.amount = 0;
            positionData.collateral = 0;

            _removeStrikeId(_strikeIds[i]);
        }
    }
```

Listing 3.1: `PutSellingVault::_settleOptions()`

Note the `CallSellingVault::_settleOptions()` routine shares a similar issue.

Recommendation Improve the above routines to allow for proper option settlement.

Status This issue has been fixed in the following commit: `664e61b`.

3.2 Improved Precision in processWithdrawalQueue()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Put/CallSellingVault`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss scenario.

In particular, we use the `PutSellingVault::processWithdrawalQueue()` as an example. This routine is used to process withdrawal requests in the pending queue. For each withdrawal request, for the given `withdrawnTokens`, we notice the current logic computes the `susdToReturn` amount as follows: `susdToReturn = current.withdrawnTokens.mulWadDown(tokenPrice)`. For improved precision, the amount can be revised as `susdToReturn = current.withdrawnTokens.mulWadUp(tokenPrice)`.

```
    function processWithdrawalQueue(uint256 idCount) external nonReentrant {
        for (uint256 i = 0; i < idCount; i++) {
            uint256 tokenPrice = getTokenPrice();

            QueuedWithdraw storage current = withdrawalQueue[queuedWithdrawalHead];

            if (block.timestamp < current.requestedTime + minWithdrawDelay) {
                return;
            }

            uint256 availableFunds = totalFunds - usedFunds;

            if (availableFunds == 0) {
                return;
            }

            uint256 susdToReturn = current.withdrawnTokens.mulWadDown(tokenPrice);
            ...
        }
```

```
    }
```

Listing 3.2: `PutSellingVault::processWithdrawalQueue()`

Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible. Note other routines share the same issue, including `CallSellingVault::processWithdrawalQueue()`, `CallSellingVault::_closeShortPosition()`, and `PutSellingVault::_closeShortPosition()`.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the following commit: `664e61b`.

3.3 Revisited StrikeId Removal in PutSellingVault

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Put/CallSellingVault`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In `Polynomial Earn`, each `vault` may manage multiple `strikes` and each `strike` has its unique `strikeId`. And the set of active `strikeId`s is managed in a storage array `liveStrikes`. While analyzing the logic to add a new `strikeId` into `liveStrikes` or remove an existing one, we notice the current removal logic can be improved.

To elaborate, we show below the `_removeStrikeId()` function. It has a rather straightforward logic in locating the index of the to-be-removed `strikeId` and then switching the located index with the last element in the array. It comes to our attention it also overwrites the last element (line 820) before immediately popping out the last element (line 821). Since the last element is immediately popped out, there is no need to overwrite it in the first place. Note it also affects the same function from the `PutSellingVault` contract.

```
809     function _removeStrikeId(uint256 _strikeId) internal {
810         uint256 i;
811         uint256 n = liveStrikes.length;
812         for (i = 0; i < n; i++) {
813             if (_strikeId == liveStrikes[i]) {
814                 break;
815             }
816         }
```

```

        if (i < n) {
            liveStrikes[i] = liveStrikes[n - 1];
            liveStrikes[n - 1] = _strikeId;
            liveStrikes.pop();
        }
    }
```

Listing 3.3: `CallSellingVault::_removeStrikeId()`

Recommendation Simplify the above `_removeStrikeId()` as follows:

```
    function _removeStrikeId(uint256 _strikeId) internal {
        uint256 i;
        uint256 n = liveStrikes.length;
        for (i = 0; i < n; i++) {
            if (_strikeId == liveStrikes[i]) {
                break;
            }
        }

        if (i < n) {
            liveStrikes[i] = liveStrikes[n - 1];
            liveStrikes.pop();
        }
    }
```

Listing 3.4: Revised `CallSellingVault::_removeStrikeId()`

Status This issue has been fixed in the following commit: `664e61b`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Polynomial Earn` protocol feature, there are privileged accounts (`owner` and `Auth`) who play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and option selling). 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 the related privileged accesses in current contracts.

```
    /// @notice Set Synthetix Volume Program Tracking Code
    /// @param _code New tracking code
    function setSynthetixTracking(bytes32 _code) external requiresAuth {
        emit UpdateSynthetixTrackingCode(synthetixTrackingCode, _code);
        synthetixTrackingCode = _code;
    }

    /// @notice Set Minimum deposit amount
    /// @param _minAmt Minimum deposit amount
    function setMinDepositAmount(uint256 _minAmt) external requiresAuth {
        emit UpdateMinDeposit(minDepositAmount, _minAmt);
        minDepositAmount = _minAmt;
    }

    /// @notice Set Deposit and Withdrawal delays
    /// @param _depositDelay New Deposit Delay
    /// @param _withdrawDelay New Withdrawal Delay
    function setDelays(uint256 _depositDelay, uint256 _withdrawDelay) external
        requiresAuth {
        emit UpdateDelays(minDepositDelay, _depositDelay, minWithdrawDelay,
            _withdrawDelay);
        minDepositDelay = _depositDelay;
        minWithdrawDelay = _withdrawDelay;
    }
```

Listing 3.5: Example Privileged Operations in `CallSellingVault`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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. The team confirms that a multi-sig account will be used to perform these privileged actions.

4 Conclusion

In this audit, we have analyzed the `Polynomial Earn (v2)` design and implementation. The `Polynomial Earn` is designed to receive asset from depositors and invest its full asset in a so-called weekly options strategy. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.