SMART CONTRACT AUDIT REPORT

for

PolyJeff

Prepared By: Xiaomi Huang

PeckShield

December 16, 2024

Document Properties

Client	PolyJeff
Title	Smart Contract Audit Report
Target	PolyJeff
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 16, 2024	Xuxian Jiang	Final Release
1.0-rc	December 15, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `PolyJeff` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About PolyJeff

`PolyJeff` is the official mascot of `Polygon` and the first hodl-to-earn memecoin on the `Polygon` network. It works is as follows: Buy any amount of `$JEFF` tokens to earn $10x$ `$JEFF` vesting tokens over a period of 10 years. Be aware that selling or transferring even a single token will result in all remaining vesting tokens being burned. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of PolyJeff

Item	Description
Name	PolyJeff
Website	https://www.polyjeff.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 16, 2024

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

- https://github.com/TechUpGroup/JEFF_SMC.git (d440170)

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

- https://github.com/TechUpGroup/JEFF_SMC.git (TBD)

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.

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `PolyJeff` 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	0	
Medium	1	■
Low	2	■ ■
Informational	0	
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 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key PolyJeff Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Front-Running/MEV For Reduced Return	Time And State	Confirmed
PVE-002	Low	Improved Constructor Logic in Pool	Coding Practices	Resolved
PVE-003	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 Possible Front-Running/MEV For Reduced Return

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

The audited protocol has a core `Pool` contract that is designed to interact with `UniswapV3 DEX` engine and manage the `DEX` liquidity. With that, it has the natural need of swapping tokens. Our analysis shows the token-swapping logic can be improved for better slippage control.

```
    function _swapTokensForTokens(uint256 tokenInAmount, address tokenIn, address
        tokenOut) private returns (uint256 tokenOutAmount) {
        IERC20(tokenIn).safeApprove(address(router), tokenInAmount);
        ISwapRouter.ExactInputSingleParams memory params =
        ISwapRouter.ExactInputSingleParams ({
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            recipient: address(this),
            deadline: block.timestamp,
            amountIn: tokenInAmount,
            amountOutMinimum: 0,
            limitSqrtPrice: 0
        });
        tokenOutAmount = router.exactInputSingleSupportingFeeOnTransferTokens(params);
    }
```

Listing 3.1: `Pool::_swapTokensForTokens()`

Specifically, if we examine the above `_swapTokensForTokens()` implementation, the helper converts the token from `tokenIn` to `tokenOut` but uses `0` as `amountOutMinimum`. As a result, the router performs the swap without the slippage control mechanism utilized. In other words, it may be sandwiched by a MEV bot for profit.

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 because the 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. The same issue is also applicable to another routine, i.e., `_swapEthForTokens()`.

Status This issue has been confirmed.

3.2 Improved Constructor Logic in Pool

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, the `Pool` contract is instantiated as a proxy with actual logic contracts in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers();`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
    function initialize(
        address _verifier,
        address _router,
        address _nonfungiblePositionManager,
```

```
        address _token,
        address _mvx,
        address _operator
    ) external initializer {
        __Pausable_init();
        __ReentrancyGuard_init();
        __Ownable_init();
        verifier = ISignatureVerifier(_verifier);
        router = ISwapRouter(_router);
        nonfungiblePositionManager = INonfungiblePositionManager(
            _nonfungiblePositionManager);
        rewardToken = IERC20(_token);
        mvxToken = IERC20(_mvx);
        operatorAddress = _operator;
        deadAddress = 0x000000000000000000000000000000000000dEaD;
        TICK_SPACING = 60;
    }
```

Listing 3.2: `Pool::initialize()`

Recommendation Improve the above-mentioned constructor routine in the `pool` contract

Status This issue has been resolved as the imported `OpenZeppelin` version does not have the `_disableInitializers()` function.

3.3 Trust Issue of Admin Keys

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

Description

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

```
    function setVerifier(address _verifier) external onlyOwner {
        verifier = ISignatureVerifier(_verifier);
    }

    function setRouter(address _router) external onlyOwner {
```

```
        router = ISwapRouter(_router);
    }

    function setNonfungiblePositionManager(address _nonfungiblePositionManager) external
         onlyOwner {
        nonfungiblePositionManager = INonfungiblePositionManager(
            _nonfungiblePositionManager);
    }

    function setOperator(address _operator) external onlyOwner {
        operatorAddress = _operator;
    }

    function setDeadAddress(address _dead) external onlyOwner {
        deadAddress = _dead;
    }

    function setTickSpacing(int24 _tickSpacing) external onlyOwner {
        require(_tickSpacing != 0, "not zero");
        TICK_SPACING = _tickSpacing;
    }

    function pause() external onlyOwner {
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }
```

Listing 3.3: Example Privileged Functions in the `Pool` Contract

If these privileged `owner` accounts are managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 as the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `PolyJeff` protocol, which is the official mascot of `Polygon` and the first hodl-to-earn memecoin on the `Polygon` network. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: 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.