SMART CONTRACT AUDIT REPORT

for

Memefi

Prepared By: Xiaomi Huang

PeckShield
May 7, 2024

Document Properties

Client	Memefi
Title	Smart Contract Audit Report
Target	Memefi
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 7, 2024	Xuxian Jiang	Final Release
1.0-rc	April 28, 2024	Xuxian Jiang	Release Candidate #1

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 design document and related source code of the Memefi 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 Memefi

Memefi is a social dapp where a user can create a KEY and others can trade it. The price of the key depends on its total supply. Each trade incurs fees to the creator, the platform, and investors. It also has a feature that allows for the distribution of a player's earnings among the key holders. When a player's reward enters the key contract, it is distributed such that a portion of the reward immediately goes to the player's address (the key creator), while the other portion is distributed among all key holders (including the creator). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Memefi

Item	Description
Name	Memefi
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 7, 2024

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

- https://github.com/memeficluborg/contracts.git (d01c0ee)

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

- https://github.com/memeficluborg/contracts.git (95078fd)

1.2 About PeckShield

PeckShield Inc. [7] 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

		Likelihood		
		High	*Medium*	*Low*
Impact	*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 OWASP Risk Rating Methodology [6]:

- 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) [5], 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 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 Logics	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 `Memefi` 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 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	

We have so far identified a list of potential issues. 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 that 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 Memefi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Mint/Burn Replay in Memefi-AssetController	Coding Practices	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Mint/Burn Replay in MemefiAssetController

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MemefiAssetController`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `Memefi` protocol has a core `MemefiAssetController` contract that is used to faciliate user mint/burn operations. While examining the logic to mint or burn, we notice the current implementation should be improved to defend against possible replay attacks.

To elaborate, we show below the related `userMint()` routine. It has a rather straightforward logic in validating the given input and then minting the requested token amount. However, when successfully validating the user input, it does not mark the given `nonce` such that it should not be used afterward. As a result, a malicious user may repeatedly use the same given input to replay the mint operation. Note the burn operation shares the same issue.

```
    function userMint(
        address to,
        address assetAddress,
        AssetType assetType,
        uint256 erc1155TokenId,
        uint256 amount,
        uint256 nonce,
        uint256 deadline,
        bytes memory signature
    ) external {
        require(deadline >= block.timestamp, "Signature expired");
        require(!isNonceUsed[nonce], "Nonce already used");
        require(amount > 0, "Amount is 0");
        require(assetAddress != address(0), "Wrong address");
        require(assetType != AssetType.Undefined, "Asset type undefined");

        bytes32 typedHash = _hashTypedDataV4(
            keccak256(
                abi.encode(
                    keccak256(
                        "Mint(address executor,address receiver,address assetAddress,
                            uint256 assetType,uint256 erc1155TokenId,uint256 amount,
                            uint256 nonce,uint256 deadline)"
                    ),
                    msg.sender,
                    to,
                    assetAddress,
                    assetType,
                    erc1155TokenId,
                    amount,
                    nonce,
                    deadline
                )
            )
        );

        require(
            ECDSA.recover(typedHash, signature) == memefiManagement.signer(),
            "Invalid signature"
        );

        uint256[] memory mintedIds = IMemefiMintableAsset(assetAddress)
            .assetsMint(to, amount, erc1155TokenId);

        emit Minted(
            nonce,
            msg.sender,
            to,
            assetAddress,
            amount,
            erc1155TokenId,
            mintedIds
        );
    }
```

Listing 3.1: `MemefiAssetController::userMint()`

Recommendation Improve the above routine by adding necessary replay defense.

Status The issue has been resolved by removing the above-mentioned functions.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```solidity
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

Because of that, a normal call to transfer() is suggested to use the safe version, i.e., safeTransfer(), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom().

In the following, we show the withdrawTokens() routine in the MemefiManagement contract. If the USDT token is supported as _token, the unsafe version of IERC20(_token).transfer(owner(), _amount) (line 176) may revert as there is no return value in the USDT token contract's transfer()/transferFrom() implementation (but the IERC20 interface expects a return value)!

```
175    function withdrawTokens(address _token, uint256 _amount) external onlyOwner {
176        IERC20(_token).transfer(owner(), _amount);
177    }
```

Listing 3.3: MemefiManagement::withdrawTokens()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom(). Other affected functions include pay(), _paymentTransfer(), and withdrawOnSwap() in MemefiKeys.

Status The issue has been resolved in the following commit: 7f54451.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In Memefi, there is a privileged administrative account, i.e., owner. The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the MemefiManagement contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
function setMemefiToken(address newMemefiToken) external onlyOwner {
    require(!memefiAddressLocked, "Memefi address locked");
    IERC20 mt = IERC20(memefiToken);
    uint256 balance = mt.balanceOf(memefiKeys);
    IMemefiKeys(memefiKeys).withdrawOnSwap(memefiToken, balance);
```

```
        mt.transfer(msg.sender, balance);
        IERC20 newToken = IERC20(newMemefiToken);
        newToken.transferFrom(msg.sender, memefiKeys, balance);
    }

    function setFeesDistributor(address _feesDistributor) external onlyOwner {
        require(feesDistributor == address(0), "Already set");
        feesDistributor = _feesDistributor;
    }

    function setMemefiKeys(address _memefiKeys) external onlyOwner {
        require(memefiKeys == address(0), "Already set");
        memefiKeys = _memefiKeys;
    }

    function revealNft(
        uint256 _revealId,
        address _erc721Contract,
        uint256 _tokenId
    ) external onlyOwner {
        _erc721revealed[_revealId][_erc721Contract][_tokenId] = true;
    }

    function setStringStorageSlot(
        uint256 _storageSlot,
        string memory _info
    ) external onlyOwner {
        _stringStorage[_storageSlot] = _info;
        emit NewStringStorageSlot(_storageSlot, _info);
    }

    function setNewTreasury(address _newTreasury) external onlyOwner {
        require(_newTreasury != address(0), "Treasury cannot be 0");
        treasury = _newTreasury;
        emit NewTreasury(_newTreasury);
    }

    function setNewSigner(address _newSigner) external onlyOwner {
        require(_newSigner != address(0), "Signer cannot be 0");
        signer = _newSigner;
        emit NewSigner(_newSigner);
    }

    function setNewRewardDistributor(
        address _newDistibutor
    ) external onlyOwner {
        require(_newDistibutor != address(0), "Reward distributor cannot be 0");
        rewardDistributor = _newDistibutor;
        emit NewRewardDistributor(_newDistibutor);
    }
```

Listing 3.4: Example Privileged Operations in `MemefiManagement`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative 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 changes 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 confirms that all the privileged roles will be transferred to a multi-sig account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Memefi protocol, which is a social dapp where a user can create a KEY and others can trade it. The price of the key depends on its total supply. Each trade incurs fees to the creator, the platform, and investors. It also has a feature that allows for the distribution of a player's earnings among the key holders. When a player's reward enters the key contract, it is distributed such that a portion of the reward immediately goes to the player's address (the key creator), while the other portion is distributed among all key holders (including the creator). 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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