



SMART CONTRACT AUDIT REPORT

for

Mu-Exchange



Prepared By: Xiaomi Huang

PeckShield
October 29, 2023

Document Properties

Client	Mu
Title	Smart Contract Audit Report
Target	Mu
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0-rc	October 29, 2023	Xuxian Jiang	Final Release
1.0-rc	October 28, 2023	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	4
1.1	About Mu	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Incorrect Fee-Handling Logic in FeeHelper	11
3.2	Improved signUp() Logic in Referrals	12
3.3	Revised Close Request Update in TradingStorage	13
3.4	Suggested Adherence of Checks-Effects-Interactions	14
3.5	Trust Issue of Admin Keys	15
4	Conclusion	18
	References	19

1 | Introduction

Given the opportunity to review the design document and related source code of the `Mu` 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 Mu

`Mu Exchange` is a decentralized perpetual exchange building on `Gnosis Chain` with the unique feature of offering leverage trading of crypto pairs with yield-bearing token (`$sDAI`) as collateral. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Mu

Item	Description
Name	Mu
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 29, 2023

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

- <https://github.com/Mu-Exchange/Mu-Contracts.git> (c3038d6)

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

- <https://github.com/Mu-Exchange/Mu-Contracts.git> (cdb006e)

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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

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.

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 \mathbb{M}_U 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	3	
Informational	0	
Total	5	

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 3 low-severity vulnerabilities.

Table 2.1: Key Mu Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Fee-Handling Logic in Fee-Helper	Business Logic	Resolved
PVE-002	Low	Improved signUp() Logic in Referrals	Coding Practices	Resolved
PVE-003	Low	Revised Close Request Update in TradingStorage	Business Logic	Resolved
PVE-004	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Resolved
PVE-005	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 Incorrect Fee-Handling Logic in FeeHelper

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FeeHelper
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To facilitate the fee collection and distribution, the `Mu` protocol provides a `FeeHelper` contract. The collected fee may be distributed to a number of entities, including `vault`, `keeper`, `treasury`, and `pool`. Our analysis on the fee-distribution logic indicates it needs to be improved.

In the following, we show the implementation of the affected `dealFee()` routine. The fee distribution to the four above-mentioned entities follows the intended design. However, the distribution to `treasury` is checked on the condition of `if(keeper != address(0) && projectFeeP > 0)` (line 101), which should be revised as `if(treasury != address(0) && projectFeeP > 0)`.

```

87     function dealFee(address trader,address keeper,uint256 totalFee,uint256 position)
           internal {
88         require(standardToken.balanceOf(address(this)) >= totalFee,"amount mismatching")
           ;
89         totalFee = dealRefereReward(totalFee,trader,position);
90         if(vault != address(0) && vaultFeeP > 0){
91             uint amount = totalFee * vaultFeeP / MILLAGE_DENOMINATOR;
92             SafeERC20.safeTransfer(standardToken,vault,amount);
93             IVault(vault).distributeReward(amount,false);
94         }
95
96         if(keeper != address(0) && orderKeeperFee > 0){
97             uint amount = totalFee * orderKeeperFee / MILLAGE_DENOMINATOR;
98             SafeERC20.safeTransfer(standardToken,keeper,amount);
99         }
100     }

```

```

101     if(keeper != address(0) && projectFeeP > 0){
102         uint amount = totalFee * projectFeeP / MILLAGE_DENOMINATOR;
103         SafeERC20.safeTransfer(standardToken,treasury,amount);
104     }
105
106     if(pool != address(0) && lpFeeP > 0){
107         uint amount = totalFee * lpFeeP / MILLAGE_DENOMINATOR;
108         standardToken.approve(pool,amount);
109         IRewardPool(pool).increaseAccTokens(amount);
110     }
111 }

```

Listing 3.1: FeeHelper::dealFee()

Recommendation Revise the above dealFee() routine to properly distribute collected fees.

Status This issue has been fixed by the following commit: 119988e.

3.2 Improved signUp() Logic in Referrals

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Referrals
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate the user referrals, the Mu protocol has a built-in Referrals contract to manage referral rewards or fee discount. In the process of analyzing the sign-up logic, we notice the current implementation can be improved.

In the following, we show the implementation of the related signUp() routine. This routine has two arguments and is defined to allow the user to register herself as a new referer. However, it comes to our attention that the first argument _user is basically not used and can be simply substituted by msg.sender.

```

236     function signUp(address _user, address _referral) public override nonReentrant {
237         require(_user != _referral, "No identical addresses");
238         require(_user != address(0), "No 0 addresses ser");
239         address user = msg.sender;
240
241         ReferrerDetails storage ref = referrerDetails[user];
242
243         uint256 referralLink = addressToUint(user);
244
245         require(ref.registered == false, "You are already registered");

```

```

246     require(
247         refLinkToUser[referralLink] == address(0),
248         "Referral Link already taken"
249     );
250
251     ref.referralLink = referralLink;
252     refLinkToUser[referralLink] = user;
253     ref.registered = true;
254
255     if (_referral == address(0)) {
256         ref.userReferredFrom = address(0);
257         referral[user] = address(0);
258         ref.canChangeReferralLink = true;
259         ref.discount = baseReferralDiscount;
260         ref.rebate = baseReferralRebate;
261     } else {
262         ReferrerDetails storage refFrom = referrerDetails[_referral];
263         require(refFrom.registered == true, "Referrer not registered");
264         ref.userReferredFrom = _referral;
265         referral[user] = _referral;
266
267         ref.canChangeReferralLink = false;
268         refFrom.userReferralList.push(user);
269         ref.discount = refFrom.discount;
270         ref.rebate = baseReferralRebate;
271     }
272     ref.tier = 1;
273     emit UserSignedUp(user, _referral);
274 }

```

Listing 3.2: Referrals::signUp()

Recommendation Revise the above routine to simplify the user sign up logic.

Status This issue has been fixed by the following commit: 119988e.

3.3 Revised Close Request Update in TradingStorage

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

Description

The Mu protocol has a dedicated TradingStorage contract to keep track of OpenTrade, penRequest, and CloseRequest. While examining current logic to store a new close request, we notice the implementation

should be improved.

In the following, we show the implementation of the affected `storeCloseRequest()` routine. It has a rather straightforward logic in storing a new `CloseRequest` and accumulating the total requested margin for closure in `Close.requestAmount` (line 130). However, it comes to our attention that it makes an update on `_request.isDone = false`, which is performed on the temporary memory variable state `_request` without being saved on the storage. To reflect the change on the storage, there is a need to relocate the update operation before the `_request` is appended to the `Close` array (line 129).

```

126     function storeCloseRequest(uint256 _orderId, CloseRequest memory _request) external
        override onlyTrading returns(uint256 index){
127         Close storage c = closes[_orderId];
128         _request.time = block.timestamp;
129         c.requests.push(_request);
130         c.requestAmount += _request.closeMargin;
131         _request.isDone = false;
132         return c.requests.length - 1 ;
133     }

```

Listing 3.3: `TradingStorage::storeCloseRequest()`

Recommendation Revise the above routine to properly store a new `CloseRequest`.

Status This issue has been fixed by the following commit: 119988e.

3.4 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MidasPair721
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [13] exploit, and the Uniswap/Lendf.Me hack [12].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the `EarnDistributor` as an example, the `claim()` function (see the code snippet below) is provided to

externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external contract (line 51) start before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

43     function claim(address _account, uint256 _amount, bytes32[] calldata _merkleProof)
44         external{
45             require(_amount > claimed[_account], "no amount to claim");
46             // Verify the merkle proof.
47             bytes32 node = keccak256(abi.encodePacked(_account, _amount));
48             require(MerkleProof.verify(_merkleProof, merkleRoot, node), "Invalid Proof");
49             uint256 amountToSend = _amount - claimed[_account];
50             require(IERC20(token).balanceOf(address(this)) >= amountToSend, "wait to deposit
51                 token");
52             IERC20(token).safeTransfer(_account, amountToSend);
53             claimed[_account] = _amount;
54             emit Claimed(_account, _amount);
55         }

```

Listing 3.4: EarnDistributor::claim()

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy, it is important to take precautions to thwart possible re-entrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle and/or utilizing the necessary `nonReentrant` modifier to block possible re-entrancy.

Status This issue has been fixed by the following commit: 119988e.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Mu` protocol, there is a privileged administrative account `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 Trading contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```

133     function setNativeFeeForKeeper(uint256 _forKeeper, uint256 _forCallback) external
        onlyOwner{
134         nativeForKeeper = _forKeeper;
135         nativeForCallback = _forCallback;
136         emit SetNativeFeeForKeeper(_forKeeper, _forCallback);
137     }
138
139     function setCallbackGasLimit(uint256 _gasLimit) external onlyOwner{
140         callbackGasLimit = _gasLimit;
141         emit SetCallbackGasLimit(_gasLimit);
142     }
143
144     function setNativeForCallback(uint256 _nativeForCallback) external onlyOwner {
145         nativeForCallback = _nativeForCallback;
146         emit SetNativeForCallback(_nativeForCallback);
147     }
148
149     function setStorage(address _tradingStorage) external onlyOwner {
150         storageT = ITradingStorage(_tradingStorage);
151         emit SetStorage(_tradingStorage);
152     }
153
154     function setStandardToken(address _standardToken) external onlyOwner {
155         standardToken = _standardToken;
156         emit SetStandardToken(_standardToken);
157     }
158
159     function setSwitch(uint256 _tradeSwitch) external onlyOwner {
160         tradeSwitch = _tradeSwitch;
161
162         emit SetSwitch(_tradeSwitch);
163     }
164
165     function setMinAcceptanceDelay(uint256 _minAcceptanceDelay) external onlyOwner {
166         minAcceptanceDelay = _minAcceptanceDelay;
167         emit MinAcceptanceDelay(_minAcceptanceDelay);
168     }

```

Listing 3.5: Example Privileged Operations in Trading

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 with the plan to transfer the privileged account to a multi-sig account.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Mu Exchange` protocol, which is a decentralized perpetual exchange. It is built on `Gnosis Chain` with the unique feature of offering leverage trading of crypto pairs with yield-bearing token (`$sDAI`) as collateral. 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-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-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.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: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.
- [12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [13] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

