SMART CONTRACT AUDIT REPORT

for

Minterest MUSDY Market

Prepared By: Xiaomi Huang

PeckShield

August 4, 2024

Document Properties

Client	Minterest
Title	Smart Contract Audit Report
Target	Minterest
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	August 4, 2024	Xuxian Jiang	Final Release
1.0-rc1	July 29, 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 smart contract source code of the MUSDY market in Minterest, 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 Minterest

Minterest is a decentralised lending protocol. It comes with a unique economic model where the protocol itself captures 100% of the value created from its functions, including interest, flash loan, and auto-liquidation fees. This value is exchanged via an automated Buyback process for its native MNT token which is then distributed to its users in return for their participation in the protocol's governance. This audit covers the support of its MUSDY market. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Minterest

Item	Description
Target	Minterest
Website	https://minterest.com/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 4, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers only the MUSDY market support, i.e., the MUSDYToken contract. With

that, we also examine related contracts that are required to support the `MUSDYToken` market, including `MToken`, `MEther` and `Supervisor`.

- https://github.com/minterest-finance/protocol.git (8397a3e)

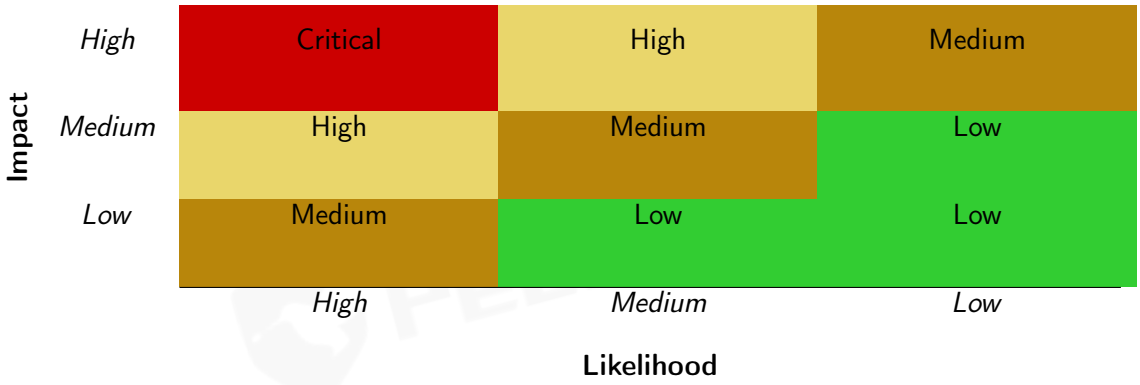
And here is the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/minterest-finance/protocol.git (d57385c2)

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



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 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 1.3: 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

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 MUSDY market in Minterest . 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 Minterest Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Precision Issue in Token::redeemFresh()/autoLiquidationSeize()	Numeric Errors	Resolved
PVE-002	Low	Inconsistent Non-Reentrancy Enforcement in Supervisor	Coding Practices	Resolved
PVE-003	Low	Improved Ether Transfer With Necessary Reentrancy Guard	Coding Practices	Resolved

Besides 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 Precision Issue in Token::redeemFresh()/autoLiquidationSeize()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `MToken`
- Category: Numeric Errors [4]
- CWE subcategory: CWE-190 [2]

Description

The `Minterest` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. While reviewing the redeem logic, we notice the current implementation has a precision issue that has been reflected in an earlier `HundredFinance` hack.

To elaborate, we show below the related `redeemFresh()` routine. As the name indicates, this routine is designed to redeem `MToken` in exchange for the underlying asset. When the user indicates the underlying asset amount (via `redeemUnderlying()`), the respective `redeemTokens` is computed as `redeemTokens = (redeemAmount * EXP_SCALE)/ exchangeRateMantissa` (line 431). Unfortunately, the current approach may unintentionally introduce a precision issue by computing the `redeemTokens` amount against the protocol. Specifically, the resulting flooring-based division introduces a precision loss, which may be just a small number but plays a critical role when certain boundary conditions are met – as demonstrated in the `HundredFinance` hack: `https://blog.hundred.finance/15-04-23-hundred-finance-hack-post-mortem-d895b618cf33`.

```
406     function redeemFresh(
407         address redeemer,
408         uint256 redeemTokens,
409         uint256 redeemAmount,
410         bool isERC20based,
411         bool isAmlProcess
```

```
412     ) internal nonReentrant returns (uint256) {
413         require(redeemTokens == 0  redeemAmount == 0, ErrorCodes.
                REDEEM_TOKENS_OR_REDEEM_AMOUNT_MUST_BE_ZERO);

415         /* exchangeRate = invoke Exchange Rate Stored() */
416         uint256 exchangeRateMantissa = exchangeRateStoredInternal();

418         if (redeemTokens > 0) {
419             /*
420              * We calculate the exchange rate and the amount of underlying to be
                    redeemed:
421              *  redeemTokens = redeemTokens
422              *  redeemAmount = redeemTokens * exchangeRateCurrent
423              */
424             redeemAmount = (redeemTokens * exchangeRateMantissa) / EXP_SCALE;
425         } else {
426             /*
427              * We get the current exchange rate and calculate the amount to be redeemed:
428              *  redeemTokens = redeemAmount / exchangeRate
429              *  redeemAmount = redeemAmount
430              */
431             redeemTokens = (redeemAmount * EXP_SCALE) / exchangeRateMantissa;
432         }
433         ...
434         return redeemAmount;
435     }
```

Listing 3.1: `MToken::redeemFresh()`

Recommendation Properly revise the above routine to ensure the precision loss needs to be computed in favor of the protocol, instead of the user. In particular, we need to ensure that markets are never empty by minting small `MToken` balances at the time of market creation so that we can prevent the rounding error being used maliciously. A deposit as small as 1 wei is sufficient. Note this issue also affects another routine, i.e., `autoLiquidationSeize()`.

Status The issue has been resolved as the team confirms that all new markets start with a zero utilization factor, then the team will provide an initial supply that will never be redeemed.

3.2 Inconsistent Non-Reentrancy Enforcement in Supervisor

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Supervisor`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

The `Minterest` protocol has a core `Supervisor` contract that mediates user operations, including `mint()/redeem()` and `borrow()/repay()`. While reviewing the mediation, we notice it can have a consistent reentrancy protection.

In the following, we examine an inconsistency among its routines, e.g., `beforeLend()` and `beforeRedeem()`. The first routine is designed to check whether the lender is allowed to proceed and the second router is used to check whether a redemption operation should be allowed. It comes to our attention that the second routine has the `nonReentrant` modifier, which is missing in the first routine. For consistency, we suggest to have the `nonReentrant` modifier in `beforeLend()` as well.

```
169     function beforeLend(IMToken mToken, address lender)
170         external
171         virtual
172         checkPausedSubject(LEND_OP, address(mToken))
173         whitelistMode(lender)
174     {
175         require(markets[mToken].isListed, ErrorCodes.MARKET_NOT_LISTED);
176         // block users from lending if AML says so
177         require(isNotBlacklisted(lender), ErrorCodes.ADDRESS_IS_BLACKLISTED);

179         // Trigger Emission system
180         rewardsHub.distributeSupplierMnt(mToken, lender);
181     }

183     /// @inheritdoc ISupervisor
184     function beforeRedeem(
185         IMToken mToken,
186         address redeemer,
187         uint256 redeemTokens,
188         bool isAmlProcess
189     ) external virtual nonReentrant checkPausedSubject(REDEEM_OP, address(mToken))
        whitelistMode(redeemer) {
190         beforeRedeemInternal(mToken, redeemer, redeemTokens, isAmlProcess);

192         // Trigger Emission system
193         rewardsHub.distributeSupplierMnt(mToken, redeemer);
```

```
194     }
```

Listing 3.2: Supervisor::beforeLend()/beforeRedeem()

Recommendation Revise the beforeLend() implementation to add the nonReentrant modifier.

Status This issue has been fixed in the following commit: d57385c2.

3.3 Improved Ether Transfer With Necessary Reentrancy Guard

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MEther
- Category: Coding Practices [3]
- CWE subcategory: CWE-1109 [1]

Description

Each asset supported by the Minterest protocol is integrated through a so-called MToken contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting MToken, users can earn interest through the MToken's exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use MTokens as collateral. There are currently two types of MTokens: MToken and MEther. While reviewing the native token support in MEther, we notice its support may be improved.

To elaborate, we show below the code snippet of the example redeemNative() routine, which allows to redeem MEther to get back the native coin (e.g., Ether). We notice that this routine directly calls the native transfer() routine (line 25) to transfer Ether. However, the transfer() is not recommended to use any more since the EIP-1884 may increase the gas cost and the 2300 gas limit may be exceeded. There is a helpful blog stop-using-solidity-transfer-now that explains why the transfer() is not recommended any more.

```
21     function redeemNative(uint256 redeemTokens) external {
22         accrueInterest();
23         uint256 redeemAmount = redeemFresh(msg.sender, redeemTokens, 0, false, false);
24         IWETH9(address(underlying)).withdraw(redeemAmount);
25         payable(msg.sender).transfer(redeemAmount);
26     }
```

Listing 3.3: MEther::redeemNative()

Recommendation Revisit the above-mentioned routine to transfer ETH using call(). Note it also affects other routines, including redeemUnderlyingNative() and borrowNative().

Status This issue has been resolved as the team confirms that these functions in `MEther` and `MMantle` are designed to improve the UX of using the protocol via the web interface. And it is not designed for use by 3rd party contracts. As a result, the recipient of `Ether` transfers in the system is always an external account (EOA), not a contract. Moreover, as low-level calls open up potential reentrancy attack possibilities, the team has decided to keep the current implementation until new requirements or use cases for this functionality will be needed.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the MUSDY market in Minterest , which is a decentralised lending protocol. It comes with a unique economic model where the protocol itself captures 100% of the value created from its functions, including interest, flash loan, and auto-liquidation fees. This value is exchanged via an automated Buyback process for its native MNT token which is then distributed to its users in return for their participation in the protocol's governance. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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 CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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