SMART CONTRACT AUDIT REPORT

for

Mori Finance

Prepared By: Xiaomi Huang

PeckShield

July 26, 2023

Document Properties

Client	Mori Finance
Title	Smart Contract Audit Report
Target	Mori Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 26, 2023	Xuxian Jiang	Final Release
1.0-rc	July 23, 2023	Xuxian Jiang	Release Candidate

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 `Mori Finance` 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 Mori Finance

`Mori Finance` is a next-generation native stable asset `DeFi` protocol built on `Ethereum`. It generates low-volatility stable assets without any loss by collateralizing `ETH`. Additionally, users have the option to create asset twins as a hedging mechanism against `ETH` price fluctuations, allowing for cost-free long positions on `ETH` without the risk of liquidation. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Mori Finance

Item	Description
Name	Mori Finance
Website	https://www.mori.finance
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 26, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this protocol assumes a trusted external oracle, which is not part of the audit.

- https://github.com/mori-defi/mori-contracts.git (4e247c6)

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

- https://github.com/mori-defi/mori-contracts.git (c5082a6)

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 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.

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

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) [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.

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 `Mori Finance` protocol, implementation. 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	1	■
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 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 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Mori Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Simplified Logic in Treasury::mint()	Coding Practices	Resolved
PVE-002	Low	Removal of Extra fToken/xToken Allowance	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Suggested Enforcement of Non-Zero Minimum Threshold	Coding Practices	Confirmed

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 Simplified Logic in Treasury::mint()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Treasury`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [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 the `Treasury` contract and notice the current arithmetic calculation may be improved.

In particular, we show below the related `Treasury::mint()` routine, which is used to calculate and mint the requested `FToken/XToken` for the given amount of base token. When the market is initialized, the current `FToken/XToken` amounts are calculated based on the `initialMintRatio` parameter (lines 273-275). We notice it involves a common factor `_totalVal`, which can be scaled down by `PRECISION` so that we can avoid the separate scale-down operation when `_fTokenOut` and `_xTokenOut` are computed respectively.

```
260  function mint(
261    uint256 _baseIn,
262    address _recipient,
263    MintOption _option
264  ) external override onlyMarket returns (uint256 _fTokenOut, uint256 _xTokenOut) {
265    StableCoinMath.SwapState memory _state = _loadSwapState();

267    if (_option == MintOption.FToken) {
268      _fTokenOut = _state.mintFToken(_baseIn);
269    } else if (_option == MintOption.XToken) {
270      _xTokenOut = _state.mintXToken(_baseIn);
```

```
    } else {
      if (_state.baseSupply == 0) {
        uint256 _totalVal = _baseIn.mul(_state.baseNav);
        _fTokenOut = _totalVal.mul(initialMintRatio).div(PRECISION).div(PRECISION);
        _xTokenOut = _totalVal.div(PRECISION).sub(_fTokenOut);
      } else {
        (_fTokenOut, _xTokenOut) = _state.mint(_baseIn);
      }
    }

    totalBaseToken = _state.baseSupply + _baseIn;

    if (_fTokenOut > 0) {
      IFractionalToken(fToken).mint(_recipient, _fTokenOut);
    }
    if (_xTokenOut > 0) {
      ILeveragedToken(xToken).mint(_recipient, _xTokenOut);
    }
  }
```

Listing 3.1: Treasury::mint()

Recommendation Simplify the above calculations to avoid duplicate arithmetic operations.

Status The issue has been fixed by this commit: `c3eaa99`.

3.2 Removal of Extra fToken/xToken Allowance

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MoriGateway`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

To facilitate the user operations, the protocol provides a `MoriGateway` contract for enhanced user experience. While examining the inherent token pre-approval logic, we notice there is redundancy in approving unnecessary tokens.

In the following, we show below the related `initialize()` routine, which, as the name indicates, is designed to initialize the `MoriGateway` contract. It comes to our attention that this gateway contract is programmed to authorize the `market` contract to move its funds, including `steth`, `fToken`, and `xToken`. Our analysis shows only `steth` authorization is needed and other two assets, i.e., `fToken` and `xToken`, do not need the pre-authorization.

```
function initialize(
  address _market,
  address _steth,
  address _fToken,
  address _xToken,
  address _pool,
  uint256 _slippage
) external initializer {
  require(_slippage <= 10000, "slippage too high");
  market = _market;
  steth = _steth;
  fToken = _fToken;
  xToken = _xToken;
  pool = _pool;
  slippage = _slippage;

  OwnableUpgradeable.__Ownable_init();
  IERC20Upgradeable(_steth).safeApprove(_pool, uint256(-1));
  IERC20Upgradeable(_steth).safeApprove(_market, uint256(-1));
  IERC20Upgradeable(_fToken).safeApprove(_market, uint256(-1));
  IERC20Upgradeable(_xToken).safeApprove(_market, uint256(-1));
}
```

Listing 3.2: `MoriGateway::initialize()`

Recommendation Revise the above token approval to ensure only minimal authorization is provided.

Status The issue has been fixed by this commit: `cb26c12`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Mori` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, adjust protocol fees, and pause/resume protocols). In the following, we show the representative functions potentially affected by the privilege of the account.

```
function updateStrategy(address _strategy) external onlyOwner {
  strategy = _strategy;
```

```

    emit UpdateStrategy(_strategy);
  }

  /// @notice Change the value of fToken beta.
  /// @param _beta The new value of beta.
  function updateBeta(uint256 _beta) external onlyOwner {
    beta = _beta;

    emit UpdateBeta(_beta);
  }

  /// @notice Change address of price oracle contract.
  /// @param _priceOracle The new address of price oracle contract.
  function updatePriceOracle(address _priceOracle) external onlyOwner {
    priceOracle = _priceOracle;

    emit UpdatePriceOracle(_priceOracle);
  }

  /// @notice Update the whitelist status for settle account.
  /// @param _account The address of account to update.
  /// @param _status The status of the account to update.
  function updateSettleWhitelist(address _account, bool _status) external onlyOwner {
    settleWhitelist[_account] = _status;

    emit UpdateSettleWhitelist(_account, _status);
  }
```

Listing 3.3: Example Privileged Operations in `Treasury`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be better if the privileged account is 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 The issue has been confirmed by the team. The team intends to manage the admin keys with a multi-sig account.

3.4 Suggested Enforcement of Non-Zero Minimum Threshold

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Market`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Mori` protocol has a core `Market` contract that allows users to mint or redeem `FToken/XToken`. The exact amount calculation is provided from a separate `StableCoinMath` algorithm. This algorithm is reminiscent of the well-known `AMM` formula. In the analysis of one example `mintFToken()`, we notice the requirement of having the non-zero base token amount. To avoid non-meaningful dust input as well as certain precision-related corner cases, we suggest the use of a dust threshold amount.

To elaborate, we show below this related routine `mintFToken()`, which has a rather straightforward logic in computing the expected `FToken` output amount from the given base token input `_baseIn`. To avoid dust input, we suggest to enforce a non-zero dust threshold amount to absorb possible precision discrepancy from the underlying `StableCoinMath` equations.

```
function mintFToken(
  uint256 _baseIn,
  address _recipient,
  uint256 _minFTokenMinted
) external override nonReentrant cachePrice returns (uint256 _fTokenMinted) {
  require(!mintPaused, "mint is paused");

  address _baseToken = baseToken;
  if (_baseIn == uint256(-1)) {
    _baseIn = IERC20Upgradeable(_baseToken).balanceOf(msg.sender);
  }
  require(_baseIn > 0, "mint zero amount");

  ITreasury _treasury = ITreasury(treasury);
  (uint256 _maxBaseInBeforeSystemStabilityMode, ) = _treasury.maxMintableFToken(
      marketConfig.stabilityRatio);

  if (fTokenMintInSystemStabilityModePaused) {
    uint256 _collateralRatio = _treasury.collateralRatio();
    require(_collateralRatio > marketConfig.stabilityRatio, "fToken mint paused");

    // bound maximum amount of base token to mint fToken.
    if (_baseIn > _maxBaseInBeforeSystemStabilityMode) {
      _baseIn = _maxBaseInBeforeSystemStabilityMode;
    }
  }
```

```

    uint256 _amountWithoutFee = _deductFTokenMintFee(_baseIn, fTokenMintFeeRatio,
        _maxBaseInBeforeSystemStabilityMode);

    IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender, address(_treasury),
        _amountWithoutFee);
    (_fTokenMinted, ) = _treasury.mint(_amountWithoutFee, _recipient, ITreasury.
        MintOption.FToken);
    require(_fTokenMinted >= _minFTokenMinted, "insufficient fToken output");

    emit Mint(msg.sender, _recipient, _baseIn, _fTokenMinted, 0, _baseIn -
        _amountWithoutFee);
  }
```

Listing 3.4: `Market::mintFToken()`

Recommendation Enforce a non-zero dust threshold amount for the base token input. The same suggestion is also applicable to another related routine, i.e., `mintXToken()`.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Mori Finance` protocol, which is a next-generation native stable asset `DeFi` protocol built on `Ethereum`. It generates low-volatility stable assets without any loss by collateralizing `ETH`. Additionally, users have the option to create asset twins as a hedging mechanism against `ETH` price fluctuations, allowing for cost-free long positions on `ETH` without the risk of liquidation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.