SMART CONTRACT AUDIT REPORT

for

Midaswap

Prepared By: Xiaomi Huang

PeckShield

June 18, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	June 18, 2023	Xuxian Jiang	Final Release
1.0-rc	June 15, 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 Midaswap 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 Midaswap

Midaswap is an NFT-liquidity protocol that innovatively uses the NFT Liquidity Book AMM algorithm to aggregate multiple liquidity providers under the same trading pair into a single liquidity pool without losing the non-fungible properties of the NFTs, thereby optimizing trading depth. Specifically, by easily adding NFT and paired token liquidity to manage market-making demand, Midaswap allows users to earn revenue from NFT transaction fees and liquidity mining. Midaswap is committed to simplifying NFT asset trading and increasing the composability of NFT assets through the integration of LP tokens, swaps, and NFT-fi, resulting in increased yields for NFT assets. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Midaswap

Item	Description
Target	Midaswap
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 18, 2023

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

- https://github.com/midaswap/midaswap-protocol-1.2.git (117e5a6b)

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

- https://github.com/midaswap/midaswap-protocol-1.2.git (a75fa3f)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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) [10], 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.

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.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Midaswap` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Undetermined	1	■
Total	5	

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, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Midaswap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Extra ETH Return in MidasRouter::addLiquidityETH()	Coding Practices	Resolved
PVE-002	Low	Incorrect NatSpec Comments in PackedUint128Math And MidasPair721	Business Logic	Resolved
PVE-003	Low	Revisited Flashloan Logic in MidasPair721::flashLoan()	Business Logic	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Undetermined	Suggested Adherence of The Checks-Effects-Interactions Pattern	Time And State	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 Extra ETH Return in MidasRouter::addLiquidityETH()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MidasRouter
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

Description

The Midaswap protocol is an innovative DEX engine that supports the trading pairs in the form of ERC721-ERC20 tokens. And liquidity providers can add the intended liquidity amount into the chosen pairs. While examining the current liquidity-adding logic, we notice the current implementation can be improved.

To elaborate, we show below the related addLiquidityETH() routine. By design, this routine adds new liquidity into the chosen pair. It has a rather straightforward logic in locating the pair contract address, computing the required Ether amount, and then performing the liquidity-adding operation. It comes to our attention that when the liquidity provider transfers extra Ether amount beyond the required amount, the extra amount will not be returned. It would be great to revisit the design by returning the extra payment, if any.

```
    function addLiquidityETH(
        address _tokenX,
        address _tokenY,
        uint24[] calldata _ids,
        uint256 _deadline
    ) external payable override returns (uint256 idAmount, uint128 lpTokenId) {
        if (_deadline < block.timestamp) revert Router__Expired();
        address _pair;
        uint256 _amount;
        _pair = factory.getPairERC721(_tokenX, _tokenY);
        _amount = _getAmountsToAdd(_pair, _ids);
        if (_tokenY != address(weth)) revert Router__WrongPair();
```

```
        if (msg.value < _amount) revert Router__WrongAmount();
        _wethDepositAndTransfer(_pair, msg.value);
        (idAmount, lpTokenId) = IMidasPair721(_pair).mintFT(_ids, msg.sender);
    }
```

Listing 3.1: `MidasRouter::addLiquidityETH()`

Recommendation Revisit the above liquidity-adding logic to return any extra liquidity amount, if any.

Status The issue has been addressed by the following commit: `eadacd0`.

3.2 Incorrect NatSpec Comments in PackedUint128Math And MidasPair721

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

The `Midaswap` protocol is well-documented with the extensive use of `NatSpec` comments to provide rich documentation for functions, return variables and others. In the process of analyzing current `NatSpec` comments, we notice the presence of numerous inconsistencies with the code implementation.

To elaborate, we show below the `findFirstLeft()` function from the `TreeMath` contract. This function is designed to identify the first id in the given tree that is higher than the given id. However, the comment indicates it is used to "Returns the first id in the tree that is higher than or equal to the given id." This comment is very misleading. And the same issue is also applicable to the `findFirstRight()` function.

```
    /**
     * @dev Returns the first id in the tree that is higher than or equal to the given
       id.
     * It will return 0 if there is no such id.
     * @param tree The tree
     * @param id The id
     * @return The first id in the tree that is higher than or equal to the given id
     */
    function findFirstLeft(
        TreeUint24 storage tree,
        uint24 id
    ) internal view returns (uint24) {
```

```
        bytes32 leaves;
        ...
    }
```

Listing 3.2: `TreeMath::findFirstLeft()`

Recommendation Remove the inconsistency among the identified misleading `NatSpec` comments. Additional inconsistencies are also present in `PackedUint128Math` and `MidasPair721` contracts.

Status The issue has been fixed by the following commits: `0fbddea` and `a75fa3f`.

3.3 Revisited Flashloan Logic in MidasPair721::flashLoan()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MidasPair721`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Midaswap` protocol provides a managing `MidasFactory721` contract to oversee the creation and management of various trading pairs. Specifically, the `MidasFactory721` contract also supports a `flashLoan()` routine (see the code snippet below). Note the `NFTs` being traded may have other rewards being attached (e.g., `ApeCoin`). Our analysis shows these rewards may need to be collected and returned back to the owner before they are traded.

As an example, we show below the related `flashLoan()` from the `MidasPair721` contract. The contract validates the flashloan caller to be `MidasFactory721` and then executes the intended flashloan operation. With that, the protocol owner is able to collect these possibly attached rewards. An improved design would allow for the callback registration to claim and send rewards back to the owner before they are sold or traded.

```
    function flashLoan(
        IMidasFlashLoanCallback receiver,
        uint256[] calldata _tokenIds,
        bytes calldata data
    ) external
        override
        nonReentrant
    {
        _checkSenderAddress(address(factory));
        uint256 length;
        length = _tokenIds.length;
        for (uint256 i; i < length; ) {
```

```
            tokenX().safeTransferFrom(
                address(this),
                address(receiver),
                _tokenIds[i]
            );
            unchecked {
                ++i;
            }
        }

        receiver.MidasFlashLoanCallback(tokenX(), _tokenIds, data);

        for (uint256 i; i < length; ) {
            if (tokenX().ownerOf(_tokenIds[i]) != address(this))
                revert MidasPair__NFTOwnershipWrong();
            unchecked {
                ++i;
            }
        }

        emit FlashLoan(msg.sender, receiver, _tokenIds);
    }
```

Listing 3.3: `MidasPair721::flashLoan()`

Recommendation Revisit the above logic to properly attribute the `NFT` rewards. The same suggestion is also applicable to the current selling/buying logic.

Status The issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Midaswap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various parameters, collect protocol fee, and adjust loyalty). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
    function setCreatePairLock(bool _newLock) external {
        require(msg.sender == owner);
```

```
        createPairLock = _newLock;
    }

    /* ========== setting parameters in Pairs ========== */

    function setRoyaltyInfo(address _tokenX, address _tokenY, bool isZero)
        external
        override
    {
        require(msg.sender == owner);
        _setRoyaltyInfo(_tokenX, _tokenY, isZero);
    }

    function setSafetyLock(address _tokenX, address _tokenY, bool _newLock) external {
        require(msg.sender == owner);
        IMidasPair721(getPairERC721[_tokenX][_tokenY]).updateSafetyLock(_newLock);
    }

    function flashLoan(
        address _tokenX,
        address _tokenY,
        IMidasFlashLoanCallback receiver,
        uint256[] calldata _tokenIds,
        bytes calldata data
    ) external {
        require(msg.sender == owner);
        IMidasPair721(getPairERC721[_tokenX][_tokenY]).flashLoan(
            receiver,
            _tokenIds,
            data
        );
    }
```

Listing 3.4: Example Privileged Operations in `MidasFactory721`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not 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 introduce `multi-sig` to mitigate this issue.

3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: MidasPair721
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

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` [14] exploit, and the `Uniswap/Lendf.Me` hack [13].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `MidasPair721` as an example, the `burn()` 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 699) 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.

```
659     function burn(
660         uint128 _LPtokenID,
661         address _nftReceiver,
662         address _to
663     )
664         external
665         override
666         // nonReentrant
667         returns (uint128 amountX, uint128 amountY) {
668         uint256[] memory _tokenIds;
669         uint256 _binIdLength;
670         uint24 originBin;
671         uint24 binStep;
672         uint128 amountFee;
673
674         _tokenIds = lpTokenAssetsMap[_LPtokenID];
675         _binIdLength = _tokenIds.length;
```

```
        (originBin, binStep, amountFee) = lpInfos[_LPtokenID].getAll();
        _checkLPTOwner(_LPtokenID, address(this));
        delete lpTokenAssetsMap[_LPtokenID];
        delete lpInfos[_LPtokenID];

        uint128 _price;
        uint24 _id;
        bytes32 _bin;
        for (uint24 i; i < _binIdLength; ) {
            unchecked {
                _id = originBin + i * binStep;
            }
            _bin = _bins[_id];
            if (_tokenIds[i] != MAX) {
                delete assetLPMap[_tokenIds[i]];

                _bin = _bin.subFirst(1e18);
                unchecked {
                    amountX += 1e18;
                }

                if (_bin.decodeX() == 0) _tree2.remove(_id);

                tokenX().safeTransferFrom(
                    address(this),
                    _nftReceiver,
                    _tokenIds[i]
                );
            } ...
        }
    }
```

Listing 3.5: `MidasPair721::burn()`

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`. Meanwhile, the `ERC721` support may naturally have the built-in support for callbacks, which deserve the special attention to guard against possible `re-entrancy`.

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

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Midaswap` protocol, which is an `NFT`-liquidity protocol (that innovatively uses the `NFT Liquidity Book AMM` algorithm to aggregate multiple liquidity providers under the same trading pair into a single liquidity pool without losing the non-fungible properties of the `NFT`s, thereby optimizing trading depth). By easily adding `NFT` and paired token liquidity to manage market-making demand, `Midaswap` allows users to earn revenue from `NFT` transaction fees and liquidity mining. `Midaswap` is committed to simplifying `NFT` asset trading and increasing the composability of `NFT` assets through the integration of `LP` tokens, swaps, and `NFT-fi`, resulting in increased yields for `NFT` assets.The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-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-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[13] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[14] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.