SMART CONTRACT AUDIT REPORT

for

NFT Protocol

Prepared By: Yiqun Chen

PeckShield

September 10, 2021

Document Properties

Client	NFT Protocol
Title	Smart Contract Audit Report
Target	NFTProtocolDEX
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 10, 2021	Xiaotao Wu	Final Release
1.0-rc1	June 25, 2021	Xiaotao Wu	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the source code of the `NFT Protocol DEX` smart contract, 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 NFT Protocol

`NFT Protocol` consists of decentralized exchange infrastructure supporting the non-fungible token (`NFT`) asset class. `NFTs` serve to represent and constitute ownership of both digital and physical assets such as digital art, in-game assets, physical art, real estate, sneakers, etc. `NFT Protocol`'s robust and all-encompassing infrastructure is intended to serve all of the needs of the `NFT` asset class and adapt to the evolving needs of the `NFT` community. The `NFT Protocol` organization is decentralized and invites collaboration from developers, entrepreneurs, and enthusiasts throughout the `NFT` sector.

The basic information of NFTProtocolDEX is as follows:

Table 1.1: Basic Information of NFTProtocolDEX

Item	Description
Name	NFT Protocol
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 10, 2021

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

- https://github.com/nftprotocol/nft-dex-audit.git (fc824b8)

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

- https://github.com/nftprotocol/nft-dex-audit.git (24949c8)

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 \ 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 the 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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 checklist of 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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 implementation of the `NFT Protocol DEX` 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	2	■■
Low	2	■■
Informational	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 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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key NFTProtocolDEX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Reentrancy Risks in make()/take()	Time and State	Fixed
PVE-002	Medium	Accommodation Of Possible Non-Compliant ERC20 Tokens	Coding Practices	Fixed
PVE-003	Low	Potential Avoidance of Fee Charge in fees()	Coding Practices	Confirmed
PVE-004	Low	Assumed Trust on Admin Keys	Security Features	Confirmed
PVE-005	Informational	Improved Ether Transfer	Business Logics	Fixed

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 Reentrancy Risks in make()/take()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Low
- Target: NFTProtocolDEX
- Category: Time and State [8]
- CWE subcategory: CWE-841 [5]

Description

In the `NFTProtocolDEX` contract, the `make()` function allows a `maker` to transfer a list of assets to the `DEX` for exchanging while the `take()` function allows a `taker` to transfer a list of expected assets to the maker's account in exchange for the maker's assets (that currently reside within the `DEX` contract), which are transferred to the taker's account.

While reviewing the make/take mechanism, we notice there are several occasions with the potential `re-entrancy` risks. Using the `make()` as an example, this function will externally call a token contract to transfer assets into the `DEX`. However, the invocation of an external contract requires extra care in avoiding the `re-entrancy` risk. The problem is essentially caused by doing `transferAssetIn()` (line 176) inside the `make()` call due to the support of ERC1155 (or similar tokens which support a callback mechanism). In this particular case, if the external contract has certain hidden logic, we may run into risk of having a `re-entrancy` via other public methods.

```
152     function make(
153         Component[] calldata _make,
154         Component[] calldata _take,
155         address[] calldata _whitelist
156     ) external payable {
157         require(!locked, "DEX shut down");
158 
159         // Prohibit multisig from making swap to maintain correct users balances
160         require(msg.sender != msig, "Multisig cannot make swap");
161 
162         // Create swap entry and transfer assets to DEX
```

```
        swap[swapsEnd].id = swapsEnd;
        swap[swapsEnd].makerAddress = msg.sender;
        require(_take.length > 0, "Empty taker array");
        for (uint256 i = 0; i < _take.length; i++) {
            checkValues(_take[i]);
            swap[swapsEnd].components[RIGHT].push(_take[i]);
        }

        // Transfer in maker assets
        uint256 totalETH;
        require(_make.length > 0, "Empty maker array");
        for (uint256 i = 0; i < _make.length; i++) {
            swap[swapsEnd].components[LEFT].push(_make[i]);
            totalETH += transferAssetIn(_make[i]);
        }
        require(msg.value >= totalETH, "Insufficient ETH");

        // Add eth to users deposited total eth balance
        usersEthBalance += msg.value;

        // Credit excess eth back to the sender
        if (msg.value > totalETH) {
            pendingWithdrawals[msg.sender] += msg.value - totalETH;
        }

        // Initialize whitelist mapping for this swap
        swap[swapsEnd].whitelistEnabled = _whitelist.length > 0;
        for (uint256 i = 0; i < _whitelist.length; i++) {
            list[swapsEnd][_whitelist[i]] = true;
        }

        // Issue event
        emit MakeSwap(_make, _take, msg.sender, _whitelist, swapsEnd);

        // Add swap
        swapsEnd += 1;
    }
```

Listing 3.1: `NFTProtocolDEX::make()`

Another similar violation can be found in the `take()` routine within the same contract.

Recommendation Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

Status The issue has been fixed by this commit: `26fe96d`.

3.2 Accommodation Of Possible Non-Compliant ERC20 Tokens

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: `NFTProtocolDEX`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [3]

Description

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

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

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

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

Listing 3.2: `ZRX.sol`

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

In the following, we show the `transfer20()` routines in the `NFTProtocolDEX` contract. If the `ZRX` token is supported as the underlying `IERC20(_comp.tokenAddress)`, the unsafe version of `coin.transferFrom(_from, _to, amount)` (line 457) may return false in the `ZRX` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a revert)! Thus, the contract has vulnerabilities against fake `transferFrom` attacks.

```
449     function transfer20(
450         Component memory _comp,
451         address _from,
452         address _to
453     ) internal {
454         checkSingleAmount(_comp);
455         IERC20 coin = IERC20(_comp.tokenAddress);
456         uint256 amount = _comp.amounts[0];
457         coin.transferFrom(_from, _to, amount);
458     }
```

Listing 3.3: `NFTProtocolDEX::transfer20()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transferFrom()`.

Status The issue has been fixed by this commit: `26fe96d`.

3.3 Potential Avoidance of Fee Charge in fees()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Medium
- Target: `NFTProtocolDEX`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [2]

Description

The `NFTProtocolDEX` smart contract implements a function `take()` that allows contract to charge a certain amount of ether as trade fee from the `taker`. The calculation of trade fee amount depends on the balance of the `taker`'s `ERC20 NFT Protocol` tokens. If the balance of the `taker` is greater than or equal to `fehi`, the trade fee amount will be 0. This can be exploited by trade `taker` to avoid trade fee.

To elaborate, we show below the `fees()` function. To avoid any fee charge, a trade `taker` may flash borrow enough `ERC20 NFT Protocol` tokens before calling `take()` and repay the borrowed tokens after calling `take()`.

```
function fees() public view returns (uint256) {
    uint256 balance = IERC20(nftProtocolTokenAddress).balanceOf(msg.sender);
    if (balance >= fehi) {
        return 0;
    }
    if (balance < felo) {
        return flat;
    }
    // Take 10% off as soon as feeBypassLow is reached
    uint256 startFee = (flat * 9) / 10;
    return startFee - (startFee * (balance - felo)) / (fehi - felo);
}
```

Listing 3.4: `NFTProtocolDEX::fees()`

Recommendation Optimize the fee charge mechanism used in the `fees()` function.

Status The issue has been confirmed. `NFT Protocol` team are aware that this is an issue at the moment, however, will not be addressing it for this release. The next version of the contract will have a proper staking mechanism to prevent accounts from taking advantage through flash borrowing.

3.4 Assumed Trust on Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `NFTProtocolDEX`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

In the `NFTProtocolDEX` contract, there is a special administrative account, i.e., `msig`. This `msig` account plays a critical role in governing and regulating the entire operation and maintenance. We examine closely the `NFTProtocolDEX` contract and identify one trust issue on this `msig` account.

To elaborate, we show below the `vote()` function. We note that the `vote()` function allows for the `msig` to update trade fee for trade `taker`.

```
    function vote(
        uint256 _flatFee,
```

```
        uint256 _feeBypassLow,
        uint256 _feeBypassHigh
    ) external {
        require(msg.sender == msig, "Unauthorized");
        require(_feeBypassLow <= _feeBypassHigh, "bypassLow must be <= bypassHigh");
        flat = _flatFee;
        felo = _feeBypassLow;
        fehi = _feeBypassHigh;
        emit Vote(_flatFee, _feeBypassLow, _feeBypassHigh);
    }
```

Listing 3.5: `NFTProtocolDEX::vote()`

We understand the need of the privileged functions for contract operation, but at the same time the extra power to the `msig` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to `msig` explicit to `NFTProtocolDEX` users.

Status The issue has been confirmed. The `msig` account was introduced to facilitate administrative tasks, e.g., withdrawing and updating trading fees. `NFT Protocol` team therefore do not interpret this issue as a concern per se. `NFT Protocol` team are aware of the risk of `msig` getting compomised and NFT Protocol losing control over fees.

3.5 Improved Ether Transfer

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `NFTProtocolDEX`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `NFTProtocolDEX` contract provides the `pull()` function for users to withdraw `Ether` funds from the contract. As for the case of transferring `Ether`, the Solidity function, `transfer()`, is used (line 296 in the code snippet below). However, as described in [1], when the `recipient` happens to be a contract which implements a callback function containing EVM instructions such as `SLOAD`, the 2300 gas supplied with `transfer()` might be insufficient, leading to an out-of-gas error.

```
289     function pull() external {
290         uint256 amount = pendingWithdrawals[msg.sender];
```

```
        pendingWithdrawals[msg.sender] = 0;
        if (msg.sender != msig) {
            // Underflow should never happen, and is handled by SafeMath if it does
            usersEthBalance -= amount;
        }
        payable(msg.sender).transfer(amount);
    }
```

Listing 3.6: `NFTProtocolDEX::pull()`

As suggested in [1], we suggest to stop using Solidity's `transfer()` as well. Note that the use of `call()` leads to side effects such as reentrancy attacks and gas token vulnerabilities.

Recommendation Replace `transfer()` with `call()`.

Status The issue has been fixed by this commit: `26fe96d`.

4 Conclusion

In this audit, we have analyzed the `NFT Protocol DEX` design and implementation. The system presents a unique, robust offering as a decentralized exchange infrastructure supporting the non-fungible token (`NFT`) asset class. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, 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] Steve Marx. Stop Using Solidity's transfer() Now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/.

[2] MITRE. CWE-1109: Use of Same Variable for Multiple Purposes. https://cwe.mitre.org/data/definitions/1109.html.

[3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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 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.