SMART CONTRACT AUDIT REPORT

for

Nested Finance V2

Prepared By: Yiqun Chen

PeckShield
October 28, 2021

Document Properties

Client	Nested Finance
Title	Smart Contract Audit Report
Target	Nested V2
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Octobor 28, 2021	Jing Wang	Final Release
1.0-rc1	Octobor 16, 2021	Jing Wang	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 design document and related smart contract source code of the Nested Finance, 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 Nested

Nested Finance is designed to be the platform with customizable financial products in the form of NFTs on decentralized protocols. In particular, the platform allows users to put several digital assets as ERC20 tokens inside an unique token called an NFT (abbreviated as NestedNFT). Each NestedNFT is backed by underlying assets, which have a real value on the market. These underlying assets are directly purchased or sold on decentralized exchanges, and stored on a self-custodian smart contract. At the end of the creation process, the user receives the NFT that encrypts every detail of his portfolio. Furthermore, Nested Finance allows users to replicate other users' NestedNFTs. The creator of the initial NestedNFTs earns royalties.

The basic information of Nested V2 is as follows:

Table 1.1: Basic Information of Nested V2

Item	Description
Name	Nested Finance
Website	https://nested.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Octobor 28, 2021

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

- https://github.com/NestedFinance/nested-core-lego.git (a9b4816)

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

- https://github.com/NestedFinance/nested-core-lego.git (1c32314)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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 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) [8], 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 1.3: 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

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 Nested V2 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	4	■ ■ ■ ■
Informational	1	■
Total	6	

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, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Nested V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistency Between Implementation and Document	Business Logic	Confirmed
PVE-002	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-004	Low	Missing Validation For _originalTokenId	Business Logic	Mitigated
PVE-005	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-006	Low	Sandwiched _calculateFees() For VIP Qualification	Business Logic	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 Inconsistency Between Implementation and Document

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Nested protocol is designed to charge 1% fees on NestedNFTs transactions. The fees are charged during the creation, update, or burn of NestedNFTs. Specifically, the NestedFactory contract takes the responsibility to collect fees by the _calculateFees() helper routine when creating or changing the orders and transfers the fees to the feeSplitter contract.

When examining the above logic, we note that the fees may be over-charged. Specifically, the _transferFeeWithRoyalty() routine accepts an argument fees with the intention to send the specific amount of tokens as the royalties. However, we notice this argument is not from the result of _calculateFees(), which charges 1% of amountSpent. The current implementation is sending the _inputTokenAmount - amountSpent amount of unspent tokens as feesAmount to the feeSplitter contract. The same issue is also present on the NestedFactory::_submitOutOrders() routine.

```
    function create(
        uint256 _originalTokenId,
        IERC20 _sellToken,
        uint256 _sellTokenAmount,
        Order[] calldata _orders
    ) external payable override nonReentrant {
        require(_orders.length > 0, "NestedFactory::create: Missing orders");

        uint256 nftId = nestedAsset.mint(msg.sender, _originalTokenId);
        (uint256 fees, IERC20 tokenSold) = _submitInOrders(nftId, _sellToken,
            _sellTokenAmount, _orders, true, false);
```

```
        _transferFeeWithRoyalty(fees, tokenSold, nftId);
        emit NftCreated(nftId, _originalTokenId);
    }
```

Listing 3.1: `NestedFactory::create()`

```
    function _submitInOrders(
        uint256 _nftId,
        IERC20 _inputToken,
        uint256 _inputTokenAmount,
        Order[] calldata _orders,
        bool _reserved,
        bool _fromReserve
    ) private returns (uint256 feesAmount, IERC20 tokenSold) {
        _inputToken = _transferInputTokens(_nftId, _inputToken, _inputTokenAmount,
            _fromReserve);
        uint256 amountSpent;
        for (uint256 i = 0; i < _orders.length; i++) {
            amountSpent += _submitOrder(_inputToken, _orders[i].token, _nftId, _orders[i
                ], _reserved);
        }
        uint256 fees = _calculateFees(msg.sender, amountSpent);
        assert(amountSpent <= _inputTokenAmount - fees); // overspent

        // If input is from the reserve, update the records
        if (_fromReserve) {
            _decreaseHoldingAmount(_nftId, address(_inputToken), _inputTokenAmount);
        }

        feesAmount = _inputTokenAmount - amountSpent;
        tokenSold = _inputToken;
    }
```

Listing 3.2: `NestedFactory::_submitInOrders()`

Moreover, there are several misleading comments embedded among current solidity code, which brings unnecessary hurdles to understand and/or maintain the software. Two example comments can be found in line 153 of `NestedRecords::updateLockTimestamp()` and line 506 of `NestedFactory::_safeTransferWithFees()`.

Using the `updateLockTimestamp()` routine as an example, the preceding function summary indicates that the new timestamp must be greater than the `block.timestamp`. However, the enforcement (lines 158 – 161) requires the new timestamp should be greater than `records[_nftId].lockTimestamp`.

```
152     /// @notice The factory can update the lock timestamp of a NFT record
153     /// The new timestamp must be greater than the block.timestamp
154     //  if block.timestamp > actual lock timestamp
155     /// @param _nftId The NFT id to get the record
156     /// @param _timestamp The new timestamp
157     function updateLockTimestamp(uint256 _nftId, uint256 _timestamp) external
            onlyFactory {
158         require(
```

```
            _timestamp > records[_nftId].lockTimestamp,
            "NestedRecords::increaseLockTimestamp: Can't decrease timestamp"
        );
        records[_nftId].lockTimestamp = _timestamp;
        emit LockTimestampIncreased(_nftId, _timestamp);
    }
```

Listing 3.3: NestedRecords::updateLockTimestamp()

```
    /// @dev Transfer from factory and collect fees (without royalties)
    /// @param _token The token to transfer
    /// @param _amount The amount (with fees) to transfer
    /// @param _dest The address receiving the funds
    function _safeTransferWithFees(
        IERC20 _token,
        uint256 _amount,
        address _dest,
        uint256 _nftId
    ) private {
        uint256 feeAmount = _calculateFees(_dest, _amount);
        _transferFeeWithRoyalty(feeAmount, _token, _nftId);
        _token.safeTransfer(_dest, _amount - feeAmount);
    }
```

Listing 3.4: NestedFactory::_safeTransferWithFees()

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The team clarifies that they do not consider the tokens not spent as "extra fees", in addition of the 1% fees. It is an economic choice to not send back the unspent tokens to the user. Also, the team has changed the logic where the 1% fee is sent with royalties and the tokens not spent are send without royalties. d269a4d.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

In the Nested protocol, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. For example, here is a payable modifier appended on the

`NestedFactory::sellTokensToNft()` function. This will make the function receive `ETHs` in an unexpected way. Note the same issue also exists on the `swapTokenForTokens()` and `sellTokensToWallet()` routines.

```
    function sellTokensToNft(
        uint256 _nftId,
        IERC20 _buyToken,
        uint256[] memory _sellTokensAmount,
        Order[] calldata _orders
    ) external payable override nonReentrant onlyTokenOwner(_nftId) isUnlocked(_nftId) {
        require(_orders.length > 0, "NestedFactory::sellTokensToNft: Missing orders");
        require(_sellTokensAmount.length == _orders.length, "NestedFactory::
            sellTokensToNft: Input lengths must match");
        require(
            nestedRecords.getAssetReserve(_nftId) == address(reserve),
            "NestedFactory::sellTokensToNft: Assets in different reserve"
        );

        (uint256 feesAmount, ) = _submitOutOrders(_nftId, _buyToken, _sellTokensAmount,
            _orders, true, true);
        _transferFeeWithRoyalty(feesAmount, _buyToken, _nftId);

        emit NftUpdated(_nftId);
    }
```

Listing 3.5: The `NestedFactory::sellTokensToNft()`

Also, the `triggerForToken()` routine performs a duplicated `approve()` (line 100) with the same function call inside `setMaxAllowance()` (line 37 and 39). We suggest to remove the redundant `approve()` for gas efficiency.

```
90      function triggerForToken(
91          bytes calldata _swapCallData,
92          address payable _swapTarget,
93          IERC20 _sellToken
94      ) external onlyOwner {
95          if (feeSplitter.getAmountDue(address(this), _sellToken) > 0) {
96              claimFees(_sellToken);
97          }

99          uint256 balance = _sellToken.balanceOf(address(this));
100         _sellToken.approve(_swapTarget, balance);
101         ExchangeHelpers.fillQuote(_sellToken, _swapTarget, _swapCallData);
102         trigger();
103         emit BuybackTriggered(_sellToken);
104     }
```

Listing 3.6: `NestedBuybacker::triggerForToken()`

```
18      function fillQuote(
19          IERC20 _sellToken,
20          address _swapTarget,
21          bytes memory _swapCallData
```

```
    ) internal returns (bool) {
        setMaxAllowance(_sellToken, _swapTarget);
        // solhint-disable-next-line avoid-low-level-calls
        (bool success, ) = _swapTarget.call(_swapCallData);
        return success;
    }
```

Listing 3.7: `ExchangeHelpers::fillQuote()`

```
    function setMaxAllowance(IERC20 _token, address _spender) internal {
        uint256 _currentAllowance = _token.allowance(address(this), _spender);
        if (_currentAllowance == 0) {
            _token.safeApprove(_spender, type(uint256).max);
        } else if (_currentAllowance != type(uint256).max) {
            _token.safeIncreaseAllowance(_spender, type(uint256).max - _currentAllowance
                );
        }
    }
```

Listing 3.8: `ExchangeHelpers::setMaxAllowance()`

Recommendation Consider the removal of the redundant code.

Status The issue has been fixed by this commit: `cc80bc0`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Nested` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show representative privileged operations in the protocol's core contract `NestedReserve` which hold all funds.

We emphasize that the privilege assignment is necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that this privileged account will be managed by a multi-sig account. Note that a compromised `owner` account would allow the attacker to change a key parameter, `factory`, which is authenticated to withdraw all the user funds from the `NestedReserve` contract.

```
    function updateFactory(address _newFactory) external onlyOwner {
        factory = _newFactory;
        emit FactoryUpdated(_newFactory);
    }
```

Listing 3.9: NestedReserve::updateFactory()

```
    function withdraw(IERC20 _token, uint256 _amount) external onlyFactory valid(address
        (_token)) {
        _token.safeTransfer(factory, _amount);
    }
```

Listing 3.10: NestedReserve::withdraw()

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies that they are going to use a multi-sig wallet for every protocol-wide operations, during phase one. In phase two, everything will be controlled by a DAO.

3.4 Missing Validation For _originalTokenId

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: NestedFactory
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the Nested protocol charges fees during the creation, update, or burn of NestedNFTs. Meanwhile, there are two ways NestedNFTs are created: built from scratch and replicated from existing NestedNFTs. At the replication time, royalties are paid to the creator of the original NestedNFTs. To elaborate, we show below the create() routine for NestedNFTs creation.

```
    function create(
        uint256 _originalTokenId,
        IERC20 _sellToken,
        uint256 _sellTokenAmount,
        Order[] calldata _orders
    ) external payable override nonReentrant {
        require(_orders.length > 0, "NestedFactory::create: Missing orders");
```

```
        uint256 nftId = nestedAsset.mint(msg.sender, _originalTokenId);
        (uint256 fees, IERC20 tokenSold) = _submitInOrders(nftId, _sellToken,
            _sellTokenAmount, _orders, true, false);

        _transferFeeWithRoyalty(fees, tokenSold, nftId);
        emit NftCreated(nftId, _originalTokenId);
    }
```

Listing 3.11: `NestedFactory::create()`

We notice that this function has an assumption that `_originalTokenId` is the replicated `NFT` id. However, there is no actual enforcement of this assumption in current implementation. The user may give arbitrary value for `_originalTokenId` and an exploiter could pass his own `NFT` id and earn the royalties without replication, which violates the design.

Recommendation Add validation for `_originalTokenId`.

Status This issue has been confirmed and partially mitigated by this commit: `4746397`.

The team clarifies they can't fully fix the issue because most of this issue is "by design" and accepted. The only parts they are fixing are:

Prevent the user from replicating a non-existent portfolio (id).

Prevent the user from replicating the portfolio created in the same transaction.

3.5 Accommodation of approve() Idiosyncrasies

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ExchangeHelpers`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

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 `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.12: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `ExchangeHelpers::setMaxAllowance()` routine as an example. This routine is designed to approve the `feeSplitter` contract to move funds on `NestedFactory`'s behalf. To accommodate the specific idiosyncrasy, for `safeIncreaseAllowance()` (line 39), there is a need to `safeApprove()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
34    function setMaxAllowance(IERC20 _token, address _spender) internal {
35        uint256 _currentAllowance = _token.allowance(address(this), _spender);
36        if (_currentAllowance == 0) {
37            _token.safeApprove(_spender, type(uint256).max);
38        } else if (_currentAllowance != type(uint256).max) {
39            _token.safeIncreaseAllowance(_spender, type(uint256).max - _currentAllowance);
40        }
41    }
```

Listing 3.13: `ExchangeHelpers::setMaxAllowance()`

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

Status The issue has been fixed by this commit: `ea598e3`.

3.6 Sandwiched _calculateFees() For VIP Qualification

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: NestedFactory
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the Nested protocol has a FeeSplitter contract that is designed to receive fees collected by the NestedFactory, and split the income among shareholders, including the NFT owners, Nested treasury and a NST buybacker contract. The fee collection gives certain discount to so-called VIP accounts. Our analysis shows the current fee collection logic of _calculateFees() can be sandwiched for VIP qualification with less fee. To elaborate, we show below the related routines for fee calculation.

```
    /// @dev Calculate the fees for a specific user and amount
    /// @param _user The user address
    /// @param _amount The amount
    /// @return The fees amount
    function _calculateFees(address _user, uint256 _amount) private view returns (
        uint256) {
        uint256 baseFee = _amount / 100;
        uint256 feeWithDiscount = baseFee - _calculateDiscount(_user, baseFee);
        return feeWithDiscount;
    }
```

Listing 3.14: NestedFactory::_calculateFees()

```
    /// @dev Calculates the discount for a VIP user
    /// @param _user User to check the VIP status of
    /// @param _amount Amount to calculate the discount on
    /// @return The discount amount
    function _calculateDiscount(address _user, uint256 _amount) private view returns (
        uint256) {
        // give a discount to VIP users
        if (_isVIP(_user)) {
            return (_amount * vipDiscount) / 1000;
        } else {
            return 0;
        }
    }
```

Listing 3.15: NestedFactory::_calculateDiscount()

```
    /// @dev Checks if a user is a VIP.
```

```
    /// User needs to have at least vipMinAmount of NST staked
    /// @param _account User address
    /// @return Boolean indicating if user is VIP
    function _isVIP(address _account) private view returns (bool) {
        if (address(smartChef) == address(0)) {
            return false;
        }
        uint256 stakedNst = smartChef.userInfo(_account).amount;
        return stakedNst >= vipMinAmount;
    }
```

Listing 3.16: `NestedFactory::_isVIP()`

We notice the `VIP` account is qualified by `vipMinAmount` amount of `NST` tokens staked into the `smartChef` contract. However, a bad actor could stake `vipMinAmount` `NST` tokens before trading with `NestedFactory` and withdraw the `NST` tokens afterwards within the same transaction.

Recommendation Take into account the `NST` staked time as well for `VIP` qualification.

Status The issue has been fixed by this commit: `1c32314`.

4 Conclusion

In this audit, we have analyzed the Nested V2 design and implementation. The system presents a unique, robust offering as a decentralized non-custodial platform with customizable financial products in the form of NFTs on decentralized protocols. 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] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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

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

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

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