SMART CONTRACT AUDIT REPORT

for

Nested Finance

Prepared By: Yiqun Chen

PeckShield

July 9, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	July 9, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 2, 2021	Xuxian Jiang	Release Candidate #1
0.3	June 19, 2021	Xuxian Jiang	Add More Findings #2
0.2	June 15, 2021	Xuxian Jiang	Add More Findings #1
0.1	June 8, 2021	Xuxian Jiang	Initial Draft

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.

The basic information of Nested is as follows:

Table 1.1: Basic Information of Nested

Item	Description
Name	Nested Finance
Website	https://nested.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 9, 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.git (a9b4816)
- https://github.com/NestedFinance/nested-token.git (687d5f3)

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.git (214e177)
- https://github.com/NestedFinance/nested-token.git (2a24d74)

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.

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 1.3: 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 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	1	■
High	0	
Medium	4	■■■■
Low	4	■■■■
Informational	0	
Total	9	

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 critical-severity vulnerability, 4 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key Nested Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Confused Spender Allowance In revokeGrant()	Business Logic	Fixed
PVE-002	Medium	Vesting Bypass With transferFrom()	Business Logic	Fixed
PVE-003	Low	Accommodation Of Possible Non-Compliant ERC20 Tokens	Coding Practices	Fixed
PVE-004	Low	Improved Sanity Checks Of System/-Function Parameters	Coding Practices	Fixed
PVE-005	Low	Possible Claim of Fee By Unrelated Entities	Business Logic	Fixed
PVE-006	Medium	Possible Royalty OverCollection In sendFeesWithRoyalties()	Business Logic	Fixed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Fixed
PVE-008	Critical	Possible Drained Reserve From exchangeAndStoreTokens()	Business Logic	Fixed
PVE-009	Low	Suggested Reentrancy Protection in NestedFactory	Time and State	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 Confused Spender Allowance In revokeGrant()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact:High
- Target: `ERC20Vestable`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Nested` protocol has its own governance token `NST` that has the flexible support with grantable ERC20 token vesting schedules. While examining the vesting schedule support, we observe an issue that may incorrectly give the grantor a wrong allowance.

To elaborate, we show below the `revokeGrant()` routine from the `ERC20Vestable` contract. This routine is designed to force a revocable grant to end based on the vested amount up to the given date. In addition, all tokens that would no longer vest are returned to the account of the original grantor. It comes to our attention that this function's caller is authenticated with `require(msg.sender == owner()|| msg.sender == grant.grantor)` (line 428).

```
422     function revokeGrant(address grantHolder) external onlyGrantor returns (bool) {
423         tokenGrant storage grant = _tokenGrants[grantHolder];
424         vestingSchedule storage vesting = _vestingSchedules[grant.vestingLocation];
425         uint256 notVestedAmount;
426 
427         // Make sure grantor can only revoke from own pool.
428         require(msg.sender == owner()  msg.sender == grant.grantor, "ERC20Vestable: not
                allowed");
429         // Make sure a vesting schedule has previously been set.
430         require(grant.isActive, "ERC20Vestable: no active vesting schedule");
431         // Make sure it's revocable.
432         require(vesting.isRevocable, "ERC20Vestable: irrevocable");
433         // Fail on likely erroneous input.
434         uint32 _today = today();
```

```
435         require(_today <= grant.startDay + vesting.duration, "ERC20Vestable: no effect")
                ;
436
437         notVestedAmount = _getNotVestedAmount(grantHolder, _today);
438
439         // Use ERC20 _approve() to forcibly approve grantor to take back not-vested
                tokens from grantHolder.
440         _approve(grantHolder, grant.grantor, notVestedAmount);
441         /* Emits an Approval Event. */
442         transferFrom(grantHolder, grant.grantor, notVestedAmount);
443         /* Emits a Transfer and an Approval Event. */
444
445         // Kill the grant
446         _tokenGrants[grantHolder] = _tokenGrants[address(0)];
447
448         emit GrantRevoked(grantHolder, _today);
449         /* Emits the GrantRevoked event. */
450         return true;
451     }
```

Listing 3.1: `ERC20Vestable::revokeGrant()`

The above logic executes as expected if the caller is the grantor. However, if it is invoked by the contract owner, the forced `_approve()` (line 440) is exercised on the `grantor`, instead of the current `msg.sender`, which may immediately fail the next `transferFrom()` statement (line 442).

Recommendation Revise the above affected routine by specifying the right approval target.

Status The issue has been fixed by this commit: `4179fdd`.

3.2 Vesting Bypass With transferFrom()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: `ERC20Vestable`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the `Nested` protocol has its own governance token `NST` with the unique support of dynamic vesting schedules. Our analysis shows that the vesting may be bypassed.

To elaborate, we show below the `transfer()/approve()` routines. Suppose there is a new grant of `100 NSTs` with the intended beneficiary `Alice` and the grant is restricted according to the specific vesting schedule. However, `Alice` can call `approve()` to allow `Malice` to spend on his behalf. With that `Malice` can immediately spend the granted `100 NSTs` without being subject to the vesting schedule.

```
    /**
     * @dev Methods transfer() and approve() require an additional available funds check
         to
     * prevent spending held but non-vested tokens. Note that transferFrom() does NOT
         have this
     * additional check because approved funds come from an already set-aside allowance,
         not from the wallet.
     */
    function transfer(address to, uint256 value) public override onlyIfFundsAvailableNow
        (msg.sender, value) returns (bool) {
        return super.transfer(to, value);
    }

    /**
     * @dev Additional available funds check to prevent spending held but non-vested
         tokens.
     */
    function approve(address spender, uint256 value) public override
        onlyIfFundsAvailableNow(msg.sender, value) returns (bool) {
        return super.approve(spender, value);
    }
```

Listing 3.2: `ERC20Vestable::transfer()/approve()`

Recommendation Properly improve the `transferFrom()` routine so that it is also restricted by the vesting schedule.

Status The issue has been fixed by this commit: `2a24d74`.

3.3 Accommodation Of Possible Non-Compliant ERC20 Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NestedFactory`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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 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 3.3: `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 `destroyForERC20()` routine in the `NestedFactory` contract. If the `USDT` token is supported as `_buyToken`, the unsafe version of `_buyToken.transfer(msg.sender, amountBought)` (line 500) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
486     /*
487     Burn NFT and Sell all tokens for a specific ERC20 then send it back to the user
488     @param  _nftId uint256 NFT token Id
489     @param _buyToken [IERC20] token used to make swaps
490     @param _swapTarget [address] the address of the contract that will swap tokens
491     @param _tokenOrders [<TokenOrder>] orders for token swaps
492     */
493     function destroyForERC20(
494         uint256 _nftId,
495         IERC20 _buyToken,
```

```
        address payable _swapTarget,
        NestedStructs.TokenOrder[] calldata _tokenOrders
    ) external onlyTokenOwner(_nftId) {
        uint256 amountBought = _destroyForERC20(_nftId, _buyToken, _swapTarget,
            _tokenOrders);
        require(_buyToken.transfer(msg.sender, amountBought), "TOKEN_TRANSFER_ERROR");
    }
```

Listing 3.4: `NestedFactory::destroyForERC20()`

Note that a similar issue but with `approve()` is present in another routine, i.e., `setMaxAllowance()`.

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

Status The issue has been fixed by the following commits: `0c0910b` and `3e9fdca`.

3.4 Improved Sanity Checks For System/Function Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Nested` protocol is no exception. Specifically, if we examine the `NestedBuybacker` contract, it has defined a number of protocol-wide risk parameters, such as `burnPercentage` and `feeSplitter`. In the following, we show the corresponding routines that allow for their changes.

```
    /**
     * @dev update the fee splitter address
     * @param _feeSplitter [address] fee splitter contract address
     */
    function setFeeSplitter(FeeSplitter _feeSplitter) public onlyOwner {
        feeSplitter = _feeSplitter;
    }

    /**
     * @dev update parts deciding what amount is sent to reserve or burned
     * @param _burnPercentage [uint] burn part
     */
    function setBurnPart(uint256 _burnPercentage) external onlyOwner {
        burnPercentage = _burnPercentage;
```

```
    }
```

Listing 3.5: A number of representative `setters` in `NestedBuybacker`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the reconfiguration logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `burnPercentage` may burn all collected fee, hence hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `34d527`.

3.5 Possible Claim of Fee By Unrelated Entities

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FeeSplitter`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

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. While examining the fee distribution logic, we notice anyone may add himself into the share recipient.

Specifically, the `FeeSplitter` contract has a `sendFeesWithRoyalties()` routine that allows to receive fees from the caller. This function accepts an argument `_royaltiesTarget` with the intention to credit the account for the royalties. However, we notice this function is permissionless so that any one can call it by specifying an arbitrary address as the `_royaltiesTarget`. While this may incur a cost for the `_royaltiesTarget` addition, the collected fee share may be sufficiently large to cover the cost.

```
    /**
     * @dev Sends a fee to this contract for splitting, as an ERC20 token
     * @param _amount [uint256] amount of token as fee to be claimed by this contract
     * @param _royaltiesTarget [address] the account that can claim royalties
     * @param _token [IERC20] currency for the fee as an ERC20 token
     * @param _nftOwner [address] user owning the NFT and paying for the fees
     */
```

```
    function sendFeesWithRoyalties(
        address _nftOwner,
        address _royaltiesTarget,
        IERC20 _token,
        uint256 _amount
    ) public {
        require(_royaltiesTarget != address(0), "FeeSplitter:
            INVALID_ROYALTIES_TARGET_ADDRESS");
        _addShares(_royaltiesTarget, _computeShareCount(_amount, royaltiesWeight,
            totalWeights), address(_token));
        _sendFees(_nftOwner, _token, _amount, totalWeights);
    }
```

Listing 3.6: `FeeSplitter::sendFeesWithRoyalties()`

Recommendation Validate the call of `sendFeesWithRoyalties()` to prevent arbitrary addition of any address to share the collected fee.

Status The issue has been resolved. The user who attempts to add to share the collected share needs to pay certain up-front cost. And economically the user may not be able to profit from doing so.

3.6 Possible Royalty OverCollection In sendFeesWithRoyalties()

- 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 with the `VIP` accounts shows the current fee collection logic can be improved.

We use the same `sendFeesWithRoyalties()` function as an example. The `VIP` account is only validated within the `sendFees()` helper routine. However, the `_royaltiesTarget` share is collected without taking into account the `VIP` discount. The no-consideration of `VIP` discount may make the internal accounting inaccurate.

```
    /**
     * @dev Sends a fee to this contract for splitting, as an ERC20 token
```

```
     * @param _amount [uint256] amount of token as fee to be claimed by this contract
     * @param _royaltiesTarget [address] the account that can claim royalties
     * @param _token [IERC20] currency for the fee as an ERC20 token
     * @param _nftOwner [address] user owning the NFT and paying for the fees
     */
    function sendFeesWithRoyalties(
        address _nftOwner,
        address _royaltiesTarget,
        IERC20 _token,
        uint256 _amount
    ) public {
        require(_royaltiesTarget != address(0), "FeeSplitter:
            INVALID_ROYALTIES_TARGET_ADDRESS");
        _addShares(_royaltiesTarget, _computeShareCount(_amount, royaltiesWeight,
            totalWeights), address(_token));
        _sendFees(_nftOwner, _token, _amount, totalWeights);
    }

    function _sendFees(
        address _nftOwner,
        IERC20 _token,
        uint256 _amount,
        uint256 _totalWeights
    ) private {
        // give a discount to VIP users
        if (_isVIP(_nftOwner)) _amount -= (_amount * vipDiscount) / 1000;
        IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);

        for (uint256 i = 0; i < shareholders.length; i++) {
            _addShares(
                shareholders[i].account,
                _computeShareCount(_amount, shareholders[i].weight, _totalWeights),
                address(_token)
            );
        }
        emit PaymentReceived(msg.sender, address(_token), _amount);
    }
```

Listing 3.7: `FeeSplitter::sendFeesWithRoyalties()`

Recommendation Take into account the `VIP` status as well for the `_royaltiesTarget` share calculation.

Status The issue has been fixed by this commit: `e8260d0`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

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 FeeSplitter contract.

```
    /**
     * @dev sets a new list of shareholders
     * @param _accounts [address[]] shareholders accounts list
     * @param _weights [uint256[]] weight for each shareholder. Determines part of the
         payment allocated to them
     */
    function setShareholders(address[] memory _accounts, uint256[] memory _weights)
        public onlyOwner {
        delete shareholders;
        require(_accounts.length > 0 && _accounts.length == _weights.length, "
            FeeSplitter: ARRAY_LENGTHS_ERR");
        totalWeights = royaltiesWeight;

        for (uint256 i = 0; i < _accounts.length; i++) {
            _addShareholder(_accounts[i], _weights[i]);
        }
    }
```

Listing 3.8: A representative setter in FeeSplitter

We emphasize that the privilege assignment is necessary and consistent with the token 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 modify a number of sensitive system parameters, which directly undermines the assumption of the Nested protocol.

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 fixed by this commit: fe1e325.

3.8 Possible Drained Reserve From exchangeAndStoreTokens()

- ID: PVE-008
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `NestedFactory`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To facilitate the token purchase and sale, the `Nested` protocol has an internal helper routine `exchangeAndStoreTokens()`. This routine is developed to purchase tokens and store them in a reserve for the user. Note this routine is used in a number of scenarios. In the following, we examine this routine and report related issues in current implementation.

To elaborate, we show below the `exchangeAndStoreTokens()` implementation. This routine allows the user to provide an arbitrary `callData` that is used directly in `address(reserve).call()` (line 211) and `ExchangeHelpers.fillQuote()` (line 215). Unfortunately, in each case, the arbitrary `callData` may be exploited to transfer all funds out of the current reserve.

```
193     function exchangeAndStoreTokens(
194         uint256 _nftId,
195         IERC20 _sellToken,
196         address payable _swapTarget,
197         NestedStructs.TokenOrder[] calldata _tokenOrders
198     ) internal {
199         uint256 buyCount = _tokenOrders.length;

201         for (uint256 i = 0; i < buyCount; i++) {
202             uint256 amountBought = 0;
203             uint256 balanceBeforePurchase = IERC20(_tokenOrders[i].token).balanceOf(
                    address(this));

205             /* If token being exchanged is the sell token, the callData sent by the
                    caller
206              ** will be used on the reserve to call the transferFromFactory, taking the
                     funds
207              ** directly instead of swapping
208              */
209             if (_tokenOrders[i].token == address(_sellToken)) {
210                 ExchangeHelpers.setMaxAllowance(_sellToken, address(reserve));
211                 (bool success, ) = address(reserve).call(_tokenOrders[i].callData);
212                 require(success, "NestedFactory: RESERVE_CALL_FAILED");
213                 amountBought = balanceBeforePurchase - IERC20(_tokenOrders[i].token).
                        balanceOf(address(this));
214             } else {
215                 bool success = ExchangeHelpers.fillQuote(_sellToken, _swapTarget,
                        _tokenOrders[i].callData);
```

```
                require(success, "NestedFactory: SWAP_CALL_FAILED");
                amountBought = IERC20(_tokenOrders[i].token).balanceOf(address(this)) - balanceBeforePurchase;
                IERC20(_tokenOrders[i].token).safeTransfer(address(reserve), amountBought);
            }
            nestedRecords.store(_nftId, _tokenOrders[i].token, amountBought, address(reserve));
        }
    }
```

Listing 3.9: `NestedFactory::exchangeAndStoreTokens()`

In particular, using the first case an example. if the `callData` is encoded to call `reserve`'s function `transfer(_recipient, _token, _amount)` where the `_amount = _token.balanceOf(reserve)`. In other words, all `_token` funds available in the reserve may be withdrawn by the current user.

Recommendation Apply necessary rigorous validity checks on untrusted user input so that the user is only allowed to access the user's funds in the reserve, not others.

Status The issue has been fixed by this commit: `b2a00d84`.

3.9 Suggested Reentrancy Protection in NestedFactory

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NestedFactory`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

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` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `NestedFactory` as an example, the `withdraw()` 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`.

Apparently, the interaction with the external contract (line 428) starts before effecting the update on internal state (line 430), 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.

```
409     /*
410     send a holding content back to the owner without exchanging it
411     @param _nftId [uint256] NFT token ID
412     @param _tokenIndex [uint256] index in array of tokens for this NFT and holding.
413     @param _token [IERC20] token address for the holding. Used to make sure previous
            index param is valid
414     */
415     function withdraw(
416         uint256 _nftId,
417         uint256 _tokenIndex,
418         IERC20 _token
419     ) external onlyTokenOwner(_nftId) {
420         require(
421             nestedRecords.getAssetTokensLength(_nftId) > _tokenIndex &&
422                 nestedRecords.getAssetTokens(_nftId)[_tokenIndex] == address(_token),
423             "INVALID_TOKEN_INDEX"
424         );
425 
426         NestedStructs.Holding memory holding = nestedRecords.getAssetHolding(_nftId,
            address(_token));
427         reserve.withdraw(IERC20(holding.token), holding.amount);
428         _transferToWallet(_nftId, holding);
429 
430         nestedRecords.deleteAsset(_nftId, _tokenIndex);
431         emit NftUpdated(_nftId, UpdateOperation.RemoveToken);
432     }
```

Listing 3.10: `NestedFactory::withdraw()`

In the meantime, we should mention that the reentrancy protection has been enforced in a number of other routines. However, it is important to take precautions in making use of `nonReentrant` for all possible functions. Note similar issues exist in other functions, including `migrateAssets()` and the adherence of `checks-effects-interactions` best practice is strongly recommended.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `29283b0`.

4 Conclusion

In this audit, we have analyzed the Nested 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.

PeckShield

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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

PeckShield

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

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

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

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

PeckShield