SMART CONTRACT AUDIT REPORT

for

NEST V3.6

Prepared By: Shuxiao Wang

PeckShield

April 20, 2021

Document Properties

Client	NEST Protocol
Title	Smart Contract Audit Report
Target	NEST V3.6
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Yiqun Chen, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 20, 2021	Xuxian Jiang	Final Release
1.0-rc1	April 9, 2021	Xuxian Jiang	Release Candidate #1
0.3	April 1, 2021	Xuxian Jiang	Additional Findings #2
0.2	March 31, 2021	Xuxian Jiang	Additional Findings #1
0.1	March 22, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **NEST V3.6** protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given branch of NEST V3.6 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 NEST Protocol

NEST Protocol is a distributed price oracle network on the Ethereum mainnet. Different from other oracle approaches, NEST uses a unique "quotation mining" mechanism to ensure that off-chain price facts are generated on the chain synchronously, and `NEST-Price` price data is directly generated on the chain, which solves the industry problem of lack of price facts on the blockchain. The `NEST-Price` price data has the characteristics of authenticity, timeliness, security, stability, etc., which can be directly referenced by the DeFi project. The audited NEST V3.6 Protocol provides a number of enhancements from earlier versions, including gas optimization, built-in redeem support, as well as voting for improved decentralized governance.

The basic information of the NEST V3.6 protocol is as follows:

Table 1.1: Basic Information of the NEST V3.6 Protocol

Item	Description
Issuer	NEST Protocol
Website	https://nestprotocol.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 20, 2021

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

- https://github.com/NEST-Protocol/NEST-Oracle-V3.6.git (e14fd5c)

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

- https://github.com/NEST-Protocol/NEST-Oracle-V3.6.git (a4bb514)

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [13]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of NEST V3.6 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	
Medium	2	
Low	5	
Informational	1	
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 high-severity vulnerability, 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key NEST V3.6 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper NTokenTags Return in NTokenControler::list()	Coding Practices	Fixed
PVE-002	Medium	Proper Calculation Of NestLedger::carveReward()	Business Logic	Fixed
PVE-003	Informational	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-004	High	Improper CarveReward Collection In NestMining	Business Logic	Fixed
PVE-005	Low	Incomplete/Redundant Logic In withdraw()	Coding Practices	Fixed
PVE-006	Low	Consistent Handling Of NEST_TOKEN_ADDRESS	Coding Practices	Confirmed
PVE-007	Low	Improved Precision By Multiplication And Division Reordering	Coding Practices	Fixed
PVE-008	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Fixed
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	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 Proper NTokenTags Return in NTokenControler::list()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NTokenControler`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

In NEST V3.6, the `NTokenControler` contract manages the instantiated `NTokens`, especially the mapping from a `token` to the corresponding `ntoken`. It also provides a number of public `setter` functions, i.e., `open()` and `setNTokenMapping()`, that can be used to create or update a `token-ntoken` mapping. In the meantime, a number of `getter` routines are also provided, e.g., `getTokenAddress()` and `getNTokenAddress()`.

In the following, we review a specific `getter` routine, i.e., `list()`, and show below its implementation. We notice that the current implementation is flawed in not advancing the index variable (line 227). In other words, the current statement of `result[i++] = nTokenTagList[index]` needs to be revised as `result[i++] = nTokenTagList[index++]`

```
206    function list(uint offset, uint count, uint order) override external view returns (
           NTokenTag[] memory) {
207
208        NTokenTag[] storage nTokenTagList = _nTokenTagList;
209        NTokenTag[] memory result = new NTokenTag[](count);
210        uint i = 0;
211
212        // reverse order
213        if (order == 0) {
214
215            uint index = nTokenTagList.length - offset;
216            uint end = index - count;
217            while (index > end) {
```

```
                result[i++] = nTokenTagList[--index];
            }
        }
        // normal order
        else {

            uint index = offset;
            uint end = index + count;
            while (index < end) {
                result[i++] = nTokenTagList[index];
            }
        }

        return result;
    }
```

Listing 3.1: NTokenControler::list()

Recommendation Revise the `list()` routine to properly return the requested `NTokenTag`s.

Status This issue has been fixed in this commit: `a4bb514`.

3.2 Proper Calculation Of NestLedger::carveReward()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `NestLedger`
- Category: Business Logic [9]
- CWE subcategory: CWE-754 [6]

Description

In NEST V3.6, there is a core `NestLedger` contract. As the name indicates, this contract maintains the protocol-wide ledger, including various `NEST/NToken/DAO` reward balances. Moreover, a number of state-changing public functions are provided. For example, the `carveReward()` function is provided to collect and distribute the deposit fee between `NEST` and `NToken` rewards. The `addReward()` function updates the new reward balance of the intended `NEST` or `NToken`, not both.

During the analysis of this contract, we notice the `carveReward()` function applies the wrong scales for distribution between `NEST` and `NToken` categories. To elaborate, we show below its implementation. It comes to our attention that the `NEST` portion of reward is currently computed with `config.nestRewardScale` (line 58) and the `NToken` portion is calculated with `config.ntokenRedardScale` (line 59). It seems these two reward scales should be exchanged for proper reward allocation.

```
49    /// @dev Carve reward
```

```
    /// @param ntokenAddress Destination ntoken address
    function carveReward(address ntokenAddress) override external payable {

        if (ntokenAddress == NEST_TOKEN_ADDRESS) {
            _nestLedger += msg.value;
        } else {
            Config memory config = _config;
            UINT storage balance = _ntokenLedger[ntokenAddress];
            balance.value = balance.value + msg.value * uint(config.nestRewardScale) /
                10000;
            _nestLedger = _nestLedger + msg.value * uint(config.ntokenRedardScale) /
                10000;
        }
    }
```

Listing 3.2: NestLedger::carveReward()

Recommendation Apply the right reward scale to compute intended mining rewards. An example revision is shown as follows:

```
    /// @dev Carve reward
    /// @param ntokenAddress Destination ntoken address
    function carveReward(address ntokenAddress) override external payable {

        if (ntokenAddress == NEST_TOKEN_ADDRESS) {
            _nestLedger += msg.value;
        } else {
            Config memory config = _config;
            UINT storage balance = _ntokenLedger[ntokenAddress];
            balance.value = balance.value + msg.value * uint(config.ntokenRedardScale) /
                 10000;
            _nestLedger = _nestLedger + msg.value * uint(config.tokenRewardScale) /
                10000;
        }
    }
```

Listing 3.3: NestLedger::carveReward()

Status This issue has been fixed in this commit: `a4bb514`.

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `NNIncome`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

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` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `NNIncome` as an example, the `claimNest()` 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 106) starts before effecting the update on internal states (lines 107 and 110), 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 very same `claimNest()` function.

```
91      function claimNest() public noContract {
92
93          // Check balance
94          IERC20 nn = IERC20(NEST_NODE_ADDRESS);
95          uint balance = nn.balanceOf(address(tx.origin));
96          require(balance > 0, "NNIncome:!balance");
97
98          // Trigger
99          uint nestAmount = miningNest();
100         _generatedNest = _generatedNest + nestAmount;
101
102         // Calculation for current mining
103         uint subAmount = _generatedNest - _infoMapping[address(tx.origin)];
104         uint thisAmount = subAmount * balance / NEST_NODE_TOTALSUPPLY;
105
```

```
        require(IERC20(NEST_TOKEN_ADDRESS).transfer(address(tx.origin), thisAmount), "
            NNIncome:!transfer");
        _infoMapping[address(tx.origin)] = _generatedNest;

        // Update latest block number of operationed
        _latestBlock = block.number;
    }
```

Listing 3.4: NNIncome::claimNest()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation It is optional as proper `nonReentrant` modifier has been in place in public routines. In the meantime, the best practice of following the `checks-effects-interactions` pattern is still recommended.

Status This issue has been fixed in this commit: `a4bb514`.

3.4 Improper CarveReward Collection In NestMining

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `NestMining`
- Category: Business Logic [9]
- CWE subcategory: CWE-754 [6]

Description

At the core of NEST V3.6 is the `NestMining` contract that provides the main entry point for interaction with mining users. In particular, mining participants may post price quotes (via `post()/post2()` routines) as well as take orders from earlier posts (via `biteToken()/biteEth()`). Each of these calls may invoke an internal helper `_collect()` to deposit the mining dividend into the `NestLedger` contract (Section 3.2). In the following, we examine this specific helper.

To elaborate, we show below the code snippet of `_collect()` that is in charge of forwarding accumulated dividends to `NestLedger`. For gas efficiency, the routine avoids making the deposit for each mining dividend. Instead, it implements a so-called `batch settlement` scheme to collect dividends. Specifically, a risk-parameter named `COLLECT_REWARD_MASK` controls the batch size. Once the batch size is achieved, the `_collect()` helper is invoked.

```
    // Deposit the accumulated dividends into nest ledger
    function _collect(
```

```
        Config memory config,
        PriceChannel storage channel,
        address ntokenAddress,
        uint length,
        uint currentFee
    ) private returns (uint) {

        uint feeUnit = uint(config.postFeeUnit) * DIMI_ETHER;
        require(currentFee % feeUnit == 0, "NM:!fee");
        uint feeInfo = channel.feeInfo;
        uint oldFee = feeInfo & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;

        // currentFee is 0, increase no fee counter
        if (currentFee == 0) {
            // channel.feeInfo = feeInfo + (1 << 128);
            channel.feeInfo = feeInfo + 0x100000000000000000000000000000000;
        }
        // length == 255 means is time to save reward
        // currentFee != oldFee means the fee is changed, need to settle
        else if (length & COLLECT_REWARD_MASK == COLLECT_REWARD_MASK || currentFee !=
            oldFee) {
            // Save reward
            INestLedger(_nestLedgerAddress).carveReward {
                value: currentFee + oldFee * (COLLECT_REWARD_MASK - (feeInfo >> 128))
            } (ntokenAddress);
            // Update fee information
            channel.feeInfo = currentFee | (((length + 1) & COLLECT_REWARD_MASK) << 128);
        }

        // Calculate share count
        return currentFee / feeUnit;
    }
```

Listing 3.5: NestMining::_collect()

Our analysis with this helper routine exposes a flaw in current implementation. Assume an initialized state with no price posts yet, a mining user makes the first post with `currentFee=1`, followed by another post with `currentFee=2`. After the first post, the call to `_collect()` leads to `feeInfo = 1 + 1<<128` and `carveReward{value: 1}(ntokenAddress))`. Right after the second post, the call to `_collect()` results in `feeInfo = 2 + 2<<128` and `carveReward{value: 2 + 1 * (256-1)}(ntokenAddress)`. Apparently, the second post will be reverted!

Recommendation Revise the `_collect()` logic to accommodate all possible cases. An example revision is shown below:

```
    // Deposit the accumulated dividends into nest ledger
    function _collect(
        Config memory config,
        PriceChannel storage channel,
        address ntokenAddress,
        uint length,
```

```
        uint currentFee
    ) private returns (uint) {

        uint feeUnit = uint(config.postFeeUnit) * DIMI_ETHER;
        require(currentFee % feeUnit == 0, "NM:!fee");
        uint feeInfo = channel.feeInfo;
        uint oldFee = feeInfo & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;

        // currentFee is 0, increase no fee counter
        if (currentFee == 0) {
            // channel.feeInfo = feeInfo + (1 << 128);
            channel.feeInfo = feeInfo + 0x100000000000000000000000000000000;
        }
        // length == 255 means is time to save reward
        // currentFee != oldFee means the fee is changed, need to settle
        else if (length & COLLECT_REWARD_MASK == COLLECT_REWARD_MASK || currentFee !=
            oldFee) {
            // Save reward
            INestLedger(_nestLedgerAddress).carveReward {
                value: currentFee + oldFee * (length&COLLECT_REWARD_MASK - (feeInfo >>
                    128))
            } (ntokenAddress);
            // Update fee information
            channel.feeInfo = currentFee | (((length + 1) & COLLECT_REWARD_MASK) << 128);
        }

        // Calculate share count
        return currentFee / feeUnit;
    }
```

Listing 3.6: NestMining::_collect()

Status This issue has been fixed in this commit: `a4bb514`.

3.5 Incomplete/Redundant Logic In withdraw()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NestMining`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

As mentioned in Section 3.4, at the core of NEST V3.6 is the `NestMining` contract that provides the main entry point for interaction with mining users. Note the contract holds the user's locked `NEST` as well as the mining pool's `NEST` and there is a public function `withdraw()` that allows users to withdraw their locked funds.

In the following, we show the code snippet of the `withdraw()` routine. It seems the code contains incomplete code snippet at lines 1221 – 1228. Note those statements at lines 1221 – 1228 do not contain meaningful code that results in any effect. In other words, based on the design, they may need to be revised to be meaningful. Or they are suggested for removal.

```
1207     /// @dev Withdraw assets
1208     /// @param tokenAddress Destination token address
1209     /// @param value The value to withdraw
1210     function withdraw(address tokenAddress, uint value) override external {

1212         // TODO: The user's locked nest and the mining pool's nest are stored together.
                 When the nest is dug up,
1213         // the problem of taking the locked nest as the ore drawing will appear
1214         // As it will take a long time for nest to finish mining, this problem will not
                 be considered for the time being
1215         UINT storage balance = _accounts[_accountMapping[msg.sender]].balances[
                 tokenAddress];
1216         //uint balanceValue = balance.value;
1217         //require(balanceValue >= value, "NM:!balance");
1218         balance.value -= value;

1220         // ntoken mining
1221         uint ntokenBalance = INToken(tokenAddress).balanceOf(address(this));
1222         if (ntokenBalance < value) {
1223             // mining

1225             // The method INToken.mined() is renamed to INToken(tokenAddress).
                     increaseTotal() in develop branch
1226             // by chenf 2021-03-31 17:25
1227             //INToken(tokenAddress).increaseTotal(value - ntokenBalance);
1228         }

1230         TransferHelper.safeTransfer(tokenAddress, msg.sender, value);
1231     }
```

Listing 3.7: NestMining::withdraw()

Recommendation Properly revise the `withdraw()` logic to add the `ntoken` mining support.

Status This issue has been fixed in this commit: `a4bb514`.

3.6 Consistent Handling Of NEST_TOKEN_ADDRESS

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

The NEST V3.6 protocol implementation includes a number of component contracts. And these component contracts need to have a consistent setup regarding common tokens and addresses. For example, as the protocol token, the NEST token contract or NEST_TOKEN_ADDRESS is often referenced for various functionality. However, NEST_TOKEN_ADDRESS is not consistently referenced.

In particular, there are two different types on the way how NEST_TOKEN_ADDRESS is used. The first type simply defines a state variable to store the NEST token address. Examples include NestVote, NestRedeeming, and NestMapping. The second type defines NEST_TOKEN_ADDRESS as an immutable state. Examples include NestLedger, NestMining, NestRedeeming, NNIncome, and NTokenController.

To facilitate future maintenance and simplify the use of the NEST token address, we suggest to make a consistent use, in choosing one of the above two types, but not both.

Recommendation Be consistent in the definition and use of NEST_TOKEN_ADDRESS.

Status This issue has been confirmed.

3.7 Improved Precision By Multiplication And Division Reordering

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: Multiple Contracts
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [3]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `NestVote::execute()` as an example. This routine is used to execute a proposal after validating all conditions have been met.

```
218     function execute(uint index) override external noContract
219     {
220         Config memory config = _config;

222         // 1. Load proposal
223         Proposal memory p = _proposalList[index];

225         // 2. Check status
226         require (uint(p.state) == uint(PROPOSAL_STATE_PROPOSED), "NestVote:!state");
227         require (block.timestamp < uint(p.stopTime), "NestVote:!time");
228         // The target address cannot already have governance permission to prevent the
                governance permission from being covered
229         address governance = _governance;
230         require(!INestGovernance(governance).checkGovernance(p.contractAddress, 0), "
                NestVote:!governance");

232         // 3. Check the gaine rate
233         IERC20 nest = IERC20(_nestTokenAddress);

235         // Calculate the circulation of nest
236         uint nestCirculation = getNestCirculation();
237         require(uint(p.gainValue) >= nestCirculation * uint(config.acceptance) / 10000,
                "NestVote:!gainValue");

239         // 3. Temporarily grant execution permission
240         INestGovernance(governance).setGovernance(p.contractAddress, 1);

242         // 4. Execute
243         _proposalList[index].state = PROPOSAL_STATE_ACCEPTED;
244         _proposalList[index].executor = address(msg.sender);
245         IVotePropose(p.contractAddress).run();

247         // 5. Delete execution permission
248         INestGovernance(governance).setGovernance(p.contractAddress, 0);

250         // Return nest
251         nest.transfer(p.proposer, uint(p.staked));

253         emit NIPExecute(msg.sender, index);
254     }
```

Listing 3.8: NestVote::execute()

We notice the validation of a successfully passed proposal (line 237) involves mixed multiplication and devision, i.e., `require(uint(p.gainValue)>= nestCirculation * uint(config.acceptance)/ 10000)`.

For improved precision, it is better to validate the condition as follows: `require(uint(p.gainValue) *10000 >= nestCirculation * uint(config.acceptance))`. By doing so, we can simply avoid possible precision loss. We highlight that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Note the price deviation check in `NestRedeeming::redeem()` (lines 146 – 147) can be simply improved.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in this commit: `a4bb514`.

3.8 Improved Sanity Checks For System/Function Parameters

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The NEST V3.6 protocol is no exception. Specifically, if we examine the `NestLedger` contract, it has defined a number of inter-related system-wide risk parameters: `nestRewardScale` and `ntokenRewardScale`. In the following, we show the corresponding routine that allows for their changes.

```
    /// @dev Modify configuration
    /// @param config Configuration object
    function setConfig(Config memory config) override external onlyGovernance {
        _config = config;
    }
```

Listing 3.9: NestLedger::setConfig()

```
    struct Config {

        // nest reward scale(10000 based). 2000
        uint32 nestRewardScale;

        // ntoken reward scale(10000 based). 8000
        uint32 ntokenRedardScale;
    }
```

Listing 3.10: The NestLedger::Config Data Structure

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 update 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 `nestRewardScale` and `ntokenRewardScale` may improperly record internal balances in the `post()/post2()` operation, potentially resulting in bookkeep inconsistency and protocol instability. Specifically, there is an intrinsic binding of these `nestRewardScale` and `ntokenRewardScale` parameters, i.e., `require(nestRewardScale + ntokenRewardScale == 10000)`, which is not enforced by 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 This issue has been fixed in this commit: `a4bb514`.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [4]

Description

In NEST V3.6, there is a privileged account, i.e., `governance`, that plays a critical role in governing and regulating various protocol-wide operations (e.g., settings of certain risk parameters). It also has the privilege to regulate or govern the flow of assets within involved components. In the following, we show a representative privileged operation in the `NetBase` contract. This routine essentially allows the `governance` to collect all funds in current contract.

```
    /// @dev Transfer funds from current contracts
    /// @param tokenAddress Destination token address. (0 means ETH)
    /// @param to Transfer in address
    /// @param value Transfer amount
    function transfer(address tokenAddress, address to, uint value) external
        onlyGovernance {
        if (tokenAddress == address(0)) {
            //address(uint160(to)).transfer(value);
            payable(to).transfer(value);
        } else {
            TransferHelper.safeTransfer(tokenAddress, to, value);
        }
```

```
    }
```

Listing 3.11: NetBase::transfer()

Also, if we examine the `NestLedger::pay()`, which also allows the `DAO applications` to transfer all funds held in the contract.

```
    /// @dev Pay
    /// @param ntokenAddress Destination ntoken address. Indicates which ntoken to pay
        with
    /// @param tokenAddress Token address of receiving funds (0 means ETH)
    /// @param to Address to receive
    /// @param value Amount to receive
    function pay(address ntokenAddress, address tokenAddress, address to, uint value)
        override external {

        require(_applications[msg.sender] > 0, "NestLedger:!app");
        if (tokenAddress == address(0)) {
            if (ntokenAddress == NEST_TOKEN_ADDRESS) {
                _nestLedger -= value;
            } else {
                UINT storage balance = _ntokenLedger[ntokenAddress];
                balance.value = balance.value - value;
            }
            payable(to).transfer(value);
        } else {
            TransferHelper.safeTransfer(tokenAddress, to, value);
        }
    }
```

Listing 3.12: NestLedger::pay()

Apparently, the `governance` account should not be a plain EOA account. The discussion with the team indicates that a planned transition to the voting-based governance greatly alleviates this concern.

Recommendation Promptly transfer the `governance` privilege to the intended governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been resolved.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the NEST V3.6 protocol. The system presents a unique offering with a distributed price oracle network. NEST V3.6 pushes forward the current oracle front-line and presents a valuable contribution to current DeFi ecosystem. The current code base is well structured and neatly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

[6] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

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

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

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

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

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