VIDMA

EG

SMART CONTRACT AUDIT

Project: EG (NFT-Staking)
Date: April 7th, 2023

TABLE OF CONTENTS

SUMMARY

Vidma is pleased to present this audit report outlining our assessment of code, smart contracts, and other important audit insights and suggestions for management, developers, and users.

The audited scope of work includes the GGStaking smart contract. The contract is a staking contract that allows users to stake and unstake their NFTs. Additionally, users can get EG tokens, which are calculated with a formula and depend on the claim fee set. These tokens are immutable and set only in constructor. Moreover, they inherited the ERC20 standard and the NFT is ERC721 accordingly. Both standards are from OpenZeppelin library, which makes their transactions safe. The contract has an access control with Ownable contract from @openZeppelin library, which makes functions such as depositReward and setTokensInfo, unreachable for other users. In addition, user should stake 8 different types of tokens to get more eg tokens.

Vulnerability Summary

During the audit process, the Vidma. A detailed summary and the current state are displayed in the table below.

Severity of the issue	Total found	Resolved	Invalid	Unresolved
Critical	0 issues	0 issues	0 issues	0 issues
High	1 issue	0 issues	1 issue	0 issues
Medium	5 issues	3 issues	2 issues	0 issues
Low	5 issues	4 issues	1 issues	0 issues
Informational	14 issues	10 issues	4 issues	0 issues
Total	**25 issues**	**17 issues**	**8 issues**	**0 issues**

After evaluating the findings in this report and the final state after fixes, the Vidma auditors can state that the contracts are fully operational and secure. Under the given circumstances, we set the following risk level:

Vidma Points System

To set the codebase quality mark, our auditors are evaluating the initial commit given for the scope of the audit and the last commit with the fixes. This approach helps us adequately and sequentially evaluate the quality of the code. Code style, optimization of the contracts, the number of issues, and risk level of the issues are all taken into consideration. The Vidma team has developed a transparent evaluation codebase quality system presented below.

Severity of the issue	Resolved	Unresolved
Critical	1	10
High	0.8	7
Medium	0.5	5
Low	0.2	0.5
Informational	0	0.1

Please note that the points are deducted out of 100 for each and every issue on the list of findings (according to the current status of the issue). Issues marked as "not valid" are not subject to point deduction.

Codebase quality: 97.30

Evaluating the **initial commit** and the **last commit with the fixes**, Vidma audit team set the following **codebase quality** mark.

Score

Based on the **overall result of the audit** and the state of the final reviewed commit, the Vidma audit team grants the following **score**:

In addition to manual check and static analysis, the auditing team has conducted a number of integrated autotests to ensure the given codebase has an adequate performance and security level.

The test results and coverage can be found in the accompanying section of this audit report.

Please be aware that this audit does not certify the definitive reliability and security level of the contract. This document describes all vulnerabilities, typos, performance issues, and security issues found by the Vidma audit team. If the code is still under development, we highly recommend running one more audit once the code is finalized.

SCOPE OF WORK

EG

EG's mission is to leverage community action and blockchain technologies to grow a global movement that defies the status quo and makes profitability intrinsically linked to positive social impact.

Within the scope of this audit, two independent auditors thoroughly investigated the given codebase and analyzed the overall security and performance of the smart contracts.

The audit was conducted from March 23, 2023 to March 30, 2023. The outcome is disclosed in this document. The review of the fixes was finished on April 7, 2023.

The scope of work for the given audit consists of the following contract:

- GGStaking

The source code was taken from the following **source**:
https://github.com/EG-Ecosystem/GG-NFT-Staking

Initial commit submitted for the audit:
6532930875a995e3a4c357fdb3b5444967179cf5

Last commit reviewed by the auditing team:
3b7670557e26985db17c18f484698189dd5ddc41

WORKFLOW OF THE AUDITING PROCESS

Vidma audit team uses the most sophisticated and contemporary methods and well-developed techniques to ensure contracts are free of vulnerabilities and security risks. The overall workflow consists of the following phases:

Phase 1: The research phase

Research

After the Audit kick-off, our security team conducts research on the contract's logic and expected behavior of the audited contract.

Documentation reading

Vidma auditors do a deep dive into your tech documentation with the aim of discovering all the behavior patterns of your codebase and analyzing the potential audit and testing scenarios.

The outcome

At this point, the Vidma auditors are ready to kick off the process. We set the auditing strategies and methods and are prepared to conduct the first audit part.

Phase 2: Manual part of the audit

Manual check

During the manual phase of the audit, the Vidma team manually looks through the code in order to find any security issues, typos, or discrepancies with the logic of the contract. The initial commit as stated in the agreement is taken into consideration.

Static analysis check

Static analysis tools are used to find any other vulnerabilities in smart contracts that were missed after a manual check.

The outcome

An interim report with the list of issues.

Phase 3: Testing part of the audit

Integration tests

Within the testing part, Vidma auditors run integration tests using the Truffle or Hardhat testing framework. The test coverage and the test results are inserted in the accompanying section of this audit report.

The outcome

Second interim report with the list of new issues found during the testing part of the audit process.

STRUCTURE AND ORGANIZATION OF THE FINDINGS

For simplicity in reviewing the findings in this report, Vidma auditors classify the findings in accordance with the severity level of the issues. (from most critical to least critical).

All issues are marked as “Resolved” or “Unresolved”, depending on if they have been fixed by EG or not. The issues with “Not Relevant” status are left on the list of findings but are not eligible for the score points deduction.

The latest commit with the fixes reviewed by the auditors is indicated in the “Scope of Work” section of the report.

The Vidma team always provides a detailed description of the issues and recommendations on how to fix them.

Classification of found issues is graded according to 6 levels of severity described below:

Critical

The issue affects the contract in such a way that funds may be lost or allocated incorrectly, or the issue could result in a significant loss.

Example: Underflow/overflow, precisions, locked funds.

High

The issue significantly affects the ability of the contract to compile or operate. These are potential security or operational issues.

Example: Compilation errors, pausing/unpausing of some functionality, a random value, recursion, the logic that can use all gas from block (too many iterations in the loop), no limitations for locking period, cooldown, arithmetic errors which can cause underflow, etc.

Medium

The issue slightly impacts the contract's ability to operate by slightly hindering its intended behavior.

Example: Absence of emergency withdrawal of funds, using assert for parameter sanitization.

Low

The issue doesn't contain operational or security risks, but are more related to optimization of the codebase.

Example: Unused variables, inappropriate function visibility (public instead of external), useless importing of SCs, misuse or disuse of constant and immutable, absent indexing of parameters in events, absent events to track important state changes, absence of getters for important variables, usage of string as a key instead of a hash, etc.

Informational

Are classified as every point that increases onboarding time and code reading, as well as the issues which have no impact on the contract's ability to operate.

Example: Code style, NatSpec, typos, license, refactoring, naming convention (or unclear naming), layout order, functions order, lack of any type of documentation.

MANUAL REPORT

Precision loss

High | MH – 01 | Invalid

In GGStaking contract the *depositReward()* function uses the accumulated reward per share, which is stored as an integer with 12 decimal places. This could lead to precision loss if the reward amount or the number of staked tokens is too large.

Recommendation:

Add precision for calculation.

Zero address of *claimFeeWallet*

Medium | MM – 01 | Resolved

In GGStaking contract the global variable *claimFeeWallet* allowed to be address 0, when it is set in function *setClaimFeeWallet()*. At the same a require *claimFeeWallet != 0* in function *setClaimFee()* does not allow to set *claimFee* variable, if the fee wallet address is zero-address. Hence, in function *claim()* fee amount can be sent to zero-address claim fee wallet and lost for always.

Recommendation:

Add require condition for *setClaimWallet()* function and delete one from *setClaimFee()*.

```
    function setClaimFeeWallet(address _claimFeeWallet) external
onlyOwner {
        require(
            _claimFeeWallet != address(0),
            "the claimFeeWallet must have a valid address"
        );
        claimFeeWallet = _claimFeeWallet;
        emit SetClaimFeeWallet(_claimFeeWallet);
    }
```

```
function setClaimFee(uint256 _claimFee) external onlyOwner {
       require(
           _claimFee > 0 && _claimFee <= 10,
           "setClaimFee: amount should be greater than 0 and smaller
than 10"
       );
       claimFee = _claimFee;
       emit SetClaimFee(_claimFee);
   }
```

Reentrancy

Medium | MM – 02 | Resolved

In GGStaking contract the function *depositReward()* uses the function transfers tokens to the contract before updating the accumulated reward per share. This could allow an attacker to re-enter the function and claim rewards multiple times before the accumulated reward per share is updated.

Recommendation:

Use checks-effects-interactions pattern; call the transfer function after all rewards updating.

Check pending rewards

Medium | MM – 03 | Invalid

In GGStaking contract *unstake()* function is not checked if the user has any pending rewards before unstaking. If the user has pending rewards, they should first claim them before unstaking their NFTs. If the user unsatake their NFTs before claiming their rewards, they could lose their pending rewards.

Recommendation:

Add checking for pending rewards before unstake them.

Thrown error

Medium | MM – 04 | Invalid

In GGStaking contract *unstake()* function is not checked if the user's *depositNumber* is less than or equel to the *totalDepositCount* before calculating the pending rewards. This could cause the function to throw an error if the user's *depositNumber* is greater that *totalDepositCount*.

Recommendation:

Add checking for *depositNumber* and *totalDepositCount*.

Invalid event parameter

Medium | TM – 01 | Resolved

The *WithdrawUnusedRewardPot* event always takes a zero value due to a typo in the code.

Recommendation:

Store the value of *unusedRewardPot* in a temporary variable before nulling it and use the stored value in the event.

```
function withdrawUnusedRewardPot() external onlyOwner {
        require(
            unusedRewardPot > 0,
            "withdrawUnusedRewardPot: unusedRewardPot should be
greater than 0"
        );
        egToken.transfer(owner(), unusedRewardPot);
        unusedRewardPot = 0;
        emit WithdrawUnusedRewardPot(unusedRewardPot);
    }
```

Absent require for constructor parameters

Low | ML – 01 | Resolved

In GGStaking contract the constructor parameters are set without checking their addresses. For example, if the address will be zero-address all funds will be lost.

Recommendation:

Add require condition for all variables in constructor.

```
 constructor(IERC721 _nftToken, IERC20 _egToken) {
 require(_nftToken != address(0) && _egToken != address(0), “Zero
address of token”);
       nftToken = _nftToken;
       egToken = _egToken;
   }
```

Inappropriate function visibility

Low | ML – 02 | Invalid

In GGStaking smart contract the *setTokenInfo* function is only called by the contract owner, it is safe to remove the external visibility modifier and replace it with public. This can result in some gas savings.

Recommendation:

```
function setTokenInfo(
      uint256[] calldata _ids,
      uint8[] calldata _isLegendaries,
      uint8[] calldata _squadIds
  ) public onlyOwner
```

Absent indexing in parameters of events

Low | ML – 03 | Resolved

In GGStaking contract events are emitted without indexed parameter, which helps to log parameters in a transaction. The reason it is important to add indexed parameter to events is because it allows for more efficient and selective event filtering. By making certain parameters as indexed, Solidity can create an index of these values, making it easier and faster to search for events with the specific parameter value.

For example, event *Staked*, *Unstaked*, *Claim*, *SetClaimWallet* need indexed parameter for address *staker*.

Recommendation:

Add indexed parameter in events.

```
event Staked(address indexed staker, uint256[] tokenId);
   event UnStaked(address indexed staker, uint256[] tokenId);
   event Claim(address indexed staker, uint256 amount);
   event SetClaimFeeWallet(address indexed_claimFeeWallet);
```

Initialization with default values

Low | TL – 01 | Resolved

In lines 405, 406, the variable is set to 0. This is a useless step that increases the use of gas.

Recommendation:

Remove initialization of variables.

```
uint256 requireStakeLegendaryCount = 0;
uint256 requireStakeCommonNFTCount = 0;
```

Re-Audit:

Please follow the same recommendations as stated in the following issue: TL - 01.

```
uint256 requireUnStakeLegendaryCount = 0;
uint256 requireUnStakeCommonNFTCount = 0;
```

Code optimization

Low | TL – 02 | Resolved

The following line of code can be shortened

```
if (userSquadTokenFeaturesSum == userSquadTokenFeatures.length) {
        return true;
} else {
        return false;
}
```

Recommendation:

Write something like this

```
return userSquadTokenFeaturesSum == userSquadTokenFeatures.length;
```

Incorrect order of layout, functions

Informational | MI – 01 | Resolved

In GGStaking smart contract has the order of functions and declared variables not the same as in solidity documentation.

Recommendation:

Layouts, functions, and modifier order should be grouped according to Solidity documentation:

Layouts:

- interfaces;
- libraries;
- contracts.

Functions:

- *constructor*;
- *receive*;
- *fallback*;
- *external*;
- *public*;
- *internal*;
- *private*;
- *view*/*pure*;

Re-audit:

After reordering variables and functions, in new comment lines 77 and 79 should be swapped to have one style.

Add Natspeck

Informational | MI – 02 | Resolved

GGStaking smart contract has no explanation for functions and variables, that makes it harder to understand the contract's logic.

Recommendation:

Add Natspeck even for internal and private functions in all contracts that do not have it.

Naming convention

Informational | MI – 03 | Resolved

Naming conventions in Solidity are important for creating clear, understandable, and maintainable code.

Recommendation:

1) CamelCase: This convention involves starting the first word of a variable name with a lowercase letter and the first letter of each subsequent word with a capital letter, without spaces or underscores. For example, *myVariable*, *myFunction*, and *myStruct*;
2) Uppercase: This convention involves writing the entire name of the constant in uppercase letters, with words separated by underscores. For example, *MY_CONSTANT*;
3) Underscore: This convention involves separating words in a variable name with an underscore. For example, *my_variable*;
4) Prefixes: Solidity developers often use prefixes to indicate the type or purpose of a variable. For example, *is* is used to indicate a boolean value, *num* for numeric values, *str* for string values, and add for address values;
5) Verb-noun pairs: When naming functions, it is common to use a verb-noun pair to describe what the function does. For example, *transferTokens*, *approveAccess* and *setFeeRate*.

Gas optimization for *setTokenInfo()*

Informational | MI – 04 | Resolved

In GGStaking contract the ‘for’ loop is executed twice for setting the token info. This can be combined into a single loop to optimize gas usage.

In the *for* loop for checking the *_squadIds* array, the condition (*_squadIds[i] < squadTokenFeatures.length*) is used to check if the squid id is valid. However, *squadTokenFeatures.length* is hardcoded to 8 and does not match the actual length of the *squadTokenFeatures* array. This can cause the function to fail for valid squad ids. A better way to check the valid squad ids would be to use *squadTokenFeatures.contains(_squadIds[i])* instead of the hardcoded length.

Recommendation:

```
function setTokenInfo(uint256[] calldata _ids, uint8[] calldata
_isLegendaries, uint8[] calldata _squadIds) public onlyOwner {
   require(_ids.length > 0, "setTokenInfo: Empty array");
   require(_ids.length == _isLegendaries.length && _ids.length ==
_squadIds.length, "setTokenInfo: the array lengths should match");

   for (uint256 i = 0; i < _ids.length; i++) {
       require(squadTokenFeatures.contains(_squadIds[i]),
"setTokenInfo: invalid squad ID");

       TokenInfo storage tokenInfo = tokenInfos[_ids[i]];
       tokenInfo.isLegendary = _isLegendaries[i] == 0 ? false :
true;
       tokenInfo.squadId = _squadIds[i];
   }
}
```

Constant for *1e12*

Informational | MI – 05 | Resolved

In GGStaking contract the *depositReward()* function uses a lot *1e12* data, without any explanation what this variable is mean.

Recommendation:

Create and declare the global constant for *1e12*.

Storage in read-only function

Informational | MI – 06 | Resolved

Since the function *userStakedNFTs()* in GGStaking contract is read-only, it is not necessary to mark the user variable as storage. This variable can be marked as memory to reduce gas costs.

Recommendation:

Use

```
UserInfo memory user = userInfos[_user];
```

Re-Audit:

In function *getPending()*, which is also view, is used *UserInfo storage user = userInfos[_user];*, but user data is not rewrited into struct. Hence, it woulb be better to use *memory* instead of *storage* to reguce gast cost.

Gas optimization for *getPending()*

Informational | MI – 07 | Resolved

In GGStaking contract function *gasPending()* is possible an optimization – remove the conditional statement if (*user.depositNumber < totalDepositCount*) because it is always true. Since *totalDepositCount* is incremented every time a deposit is made, it will always be greater than or equal to *user.depositNumber*.

Recommendation:

```
function getPending(address _user) external view returns (uint256) {
  UserInfo storage user = userInfos[_user];
  uint256 pending = user.pendingRewards;
  if (user.depositNumber < totalDepositCount) {
     pending += _getPending(_user) - user.rewardDebt;
  }
  return pending;
}
```

Gas optimization for *unstake()*

Informational | MI – 08 | Invalid

In GGStaking contract function *unstake()* has the *lastTokenIndex* variable, which is also unnecessary and can be removed by using the length of the *ownedTokens* mapping.

The function iterates twice over the tokens array. The first iteration is to validate the tokens, and the second iteration is to unstake them. Instead, the function could combine both interactions into a single loop to reduce gas cost.

Recommendation:

Implement gas optimization advice.

Gas optimization for *stake()*

Informational | MI – 09 | Invalid

GGStaking contract function *stake()* has the use of nested for loops to check duplicates and ownership of each NFT in the input array is a costly operation, especially if the input array contains many elements.

Recommendation:

It would be more gas efficient to use a mapping to keep track of which NFTs have already been checked.

Usage of *transferFrom* inside the loop

Informational | MI – 10 | Invalid

The usage of the *transferFrom* function inside the *for* loop is not recommended (function *stake*), as it is a costly operation, and the cost increases with the number of NFTs being transferred.

Recommendation:

It would be more gas efficient to use a single *sefeBatchTransferFrom* function from the ERC721 contract to transfer all the NFTs at once.

Unnecessary checking

Informational | MI – 11 | Invalid

In GGStaking contract the function *stake* has the *if* statement checking if *requireStakeLegendaryCount > 0* is unnecessary since *requireStakeLegendaryCount* will always be greater than or equal to zero.

Recommendation:

Remove this if statement.

Replace twice call

Informational | MI – 12 | Resolved

In GGStaking contract the function *claim* has *egToken.balalnceOf(address(this))*, that call is made twice, which can be replaced with a single call and stored in a variable for efficiency.

Recommendation:

Create a variable *contractBalance = egToken.balalnceOf(address(this))* and use the variable in function.

Gas optimization for *_getPending()*

Informational | MI – 13 | Resolved

GGStaking contract function *_getPending()* has some room for gas optimization. The function performs several divisions by the constant *1e12*. These divisions can be avoided by multiplying the final result by *1e12*. Additionally, there is no need to declare the pending variable before assigning it a value, as it is only used to store intermediate values. Instead, the function can directly return the final result.

Recommendation:

```
function _getPending(address _user) public view returns (uint256) {
    UserInfo storage user = userInfos[_user];
    uint256 pending;
    if (user.isLegendaryStaker) {
      pending = user.stakedLegendaryCountForHolder *
accLegendaryPerShare;
    }
    if (user.isAllSquadStaker) {
      pending += accAllSquadPerShare;
    }
    if (user.commonNFTHolder) {
      pending += user.commonNFTCountForHolder *
accCommonNFTPerShare;
    }
    return pending / 1e12;
  }
```

Gas optimization for *checkAllSquadStaker()*

Informational | MI – 14 | Resolved

The function can be optimized by breaking the loop early if the user has already staked NFTs from all eight squads. This can be done by adding a break statement after the return true statement.

The *userSquadTokenFeaturesSum* variable can be incremented inside the loop using the *+=* operator instead of the *+* operator.

Recommendation:

```
function checkAllSquadStaker() private view returns (bool) {
  UserInfo storage user = userInfos[msg.sender];
  uint8[] memory userSquadTokenFeatures = new uint8[]
(squadTokenFeatures.length);

  for (uint256 i = 0; i < user.totalNFTCountForHolder; i++) {
    uint256 tokenId = ownedTokens[msg.sender][i];
    if (tokenId == 0) continue; // check if the tokenId is valid
    TokenInfo storage tokenInfo = tokenInfos[tokenId];
    if (tokenInfo.isLegendary) continue;
    userSquadTokenFeatures[tokenInfo.squadId] = 1;
    if (userSquadTokenFeaturesSum == userSquadTokenFeatures.length) {
      return true;
    }
  }

  for (uint8 squadId = 0; squadId < userSquadTokenFeatures.length;
squadId++) {
    if (userSquadTokenFeatures[squadId] == 0) {
      return false;
    }
  }

  return true;
}
```

Re-Audit:

Please follow the recommendations when it is better to use storage data location and when memory.

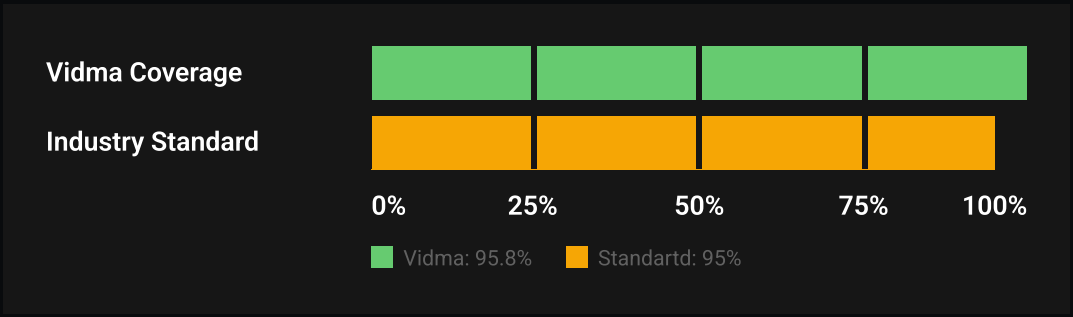
Memory, is a temporary data location used for storing variables that are only needed during the execution of a particular function. Memory is cheaper to access than storage, but its contents are lost once the function call is completed. Memory is typically used for variables that are only needed within the scope of a single function.

Another recommendation is to save gas-cost. As this contract has pragma version 0.8.17, which has internal SafeMath implementation, math operation can be used '+='. Please check all contracts for that.

TEST RESULTS

To verify the security of contracts and their performance, a number of integration tests were carried out using the Hardhat testing framework.

In this section, we provide both tests written by EG and tests written by Vidma auditors.



It is important to note that Vidma auditors do not modify, edit or add tests to the existing tests provided in the EG repository. We write totally separate tests with code coverage of a minimum of 95% to meet the industry standards.

Tests Written by Vidma Auditors

Test Coverage

File	% Stmts	% Branch	% Funcs	% Lines
contracts\	95.80	77.59	100.00	84.69
GGStaking.sol	95.80	77.59	100.00	84.69
All Files	**95.80**	**77.59**	**100.00**	**84.69**

Test Results

```
Contract: GGStaking
  initialization
    ✓ should set owner correctly (39ms)
    ✓ should set initial parameters correctly
  setClaimFee
    ✓ should revert if caller is not owner (93ms)
    ✓ should revert if the fee exceeds the limit (46ms)
    ✓ should revert if the address of the fee wallet is not set
    ✓ should set the fee correctly (70ms)
    ✓ should catch event (94ms)
  setClaimFeeWallet
    ✓ should revert if caller is not owner (62ms)
    ✓ should set the address of the fee wallet correctly (192ms)
    ✓ should catch event
  setTokenInfo
    ✓ should revert if caller is not owner (67ms)
    ✓ should revert if empty arrays are passed (39ms)
    ✓ should revert if arrays are of different lengths (46ms)
    ✓ should revert if the squadId is greater than their
      number (39ms)
    ✓ should set tokens info correctly (440ms)
  stake
```

```
    ✓ should revert if passed empty array (65ms)
    ✓ should revert if staker is not token owner (113ms)
    ✓ should revert if duplicate token ids (67ms)
    ✓ should stake NFT correctly (1800ms)
    ✓ should catch event
  setRewardsPercent
    ✓ should revert if caller is not owner
    ✓ should revert if the sum of percentages is not equal to 100
    ✓ should set reward percentages correctly (61ms)
    ✓ should catch event
  depositReward
    ✓ should revert if reward percentages is not set (169ms)
    ✓ should revert if nft tokens have not been staked before (56ms)
    ✓ should deposit correctly (90ms)
    ✓ should catch event
  getPending
    ✓ should return pending amount correctly (41ms)
  unstake
    ✓ should revert if passed empty array
    ✓ should revert if staker is not token owner (45ms)
    ✓ should revert if duplicate token ids
    ✓ should unstake NFT correctly (152ms)
  claim
    ✓ should revert if the pending amount is a 0 (112ms)
    ✓ should claim tokens correctly (102ms)
    ✓ should catch event
  withdrawUnusedRewardPot
    ✓ should withdraw correctly (92ms)
    ✓ should revert if no available tokens
    - should catch event
  userStakedNFTs
    ✓ should return staked NFTs correctly (81ms)

39 passing (6s)
1 pending
```

We are delighted to have a chance to work with the EG team and contribute to your company's success by reviewing and certifying the security of your smart contracts.

The statements made in this document should be interpreted neither as investment or legal advice, nor should its authors be held accountable for decisions made based on this document.

Vidma is a security audit company helping crypto companies ensure their code and products operate safely and as intended, enabling founders to sleep soundly at night. We specialize in auditing DeFi protocols, layer one protocols, and marketplace solutions. Our team consists of experienced and internationally trained specialists. Our company is based in Ukraine, known for its strong engineering, cryptography, and cybersecurity culture.

Website: **vidma.io**
Email: **security@vidma.io**