# invozone

# InvoAudit

## SMART CONTRACT AUDIT REPORT

for

## InvoBlox NFT Staking Contract

Prepared By: Muhammad Talha

InvoAudit

February 7, 2023

## Document Properties

| | |
|---|---|
| **Client** | InvoBlox |
| **Title** | Smart Contract Audit Report |
| **Target** | NFT Staking Contract |
| **Version** | 1.0 |
| **Author** | Muhammad Talha |
| **Auditors** | Muhammad Talha |
| **Reviewed by** | Auditing Team |
| **Approved by** | Auditing Team |
| **Classification** | Public |

## Version Properties

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | February 8, 2023 | Muhammad Talha | Final Release |
| 1.0-rc | February 1, 2023 | Muhammad Talha | Release Candidate |

## Contact

| | |
|---|---|
| **Name** | InvoAudit |
| **Phone** | +98798-4567 |
| **Email** | InvoAudits@invoblox.com |

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of Auditchain, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

## 1.1 About NFT Staking

NFT staking refers to locking up NFTs on a platform or protocol to earn rewards and other privileges. This allows NFT holders to put their idle assets to work without having to sell them.

Staking an NFT works like staking cryptocurrency; all you need is a Web3 wallet. That said, not all NFTs can be staked. If you're considering buying digital collectibles with a view to stake them, double-check that your preferred staking service supports the collection before making a purchase.

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

- https://goerli.etherscan.io/address/0x2a106267ffa2c1edea89292c2c5a7423e368ad31#code

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

- https://goerli.etherscan.io/address/0x55a65821582972f91a5b94d22b8abca7b4c9410c#code

## 1.2 About InvoAudit

InvoAudit Inc. 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).

## 1.3 Methodology

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

- 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,

  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.

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 con- tracts 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), 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.

## 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.

## 1.5 Vulnerability Classifications

| Vulnerability Classes | |
|---|---|
| **Class** | **Description** |
| Access Controls | Related to authorization of users and assessment of rights |
| Auditing and Logging | Related to auditing of actions or logging of problems |
| Authentication | Related to the identification of users |
| Configuration | Related to security configurations of servers, devices or software |
| Cryptography | Related to protecting the privacy or integrity of data |
| Data Exposure | Related to unintended exposure of sensitive information |
| Data Validation | Related to improper reliance on the structure or values of data |
| Denial of Service | Related to causing system failure |
| Error Reporting | Related to the reporting of error conditions in a secure fashion |
| Patching | Related to keeping software up to date |
| Session Management | Related to the identification of authenticated users |
| Timing | Related to race conditions, locking or order of operations |
| Undefined Behavior | Related to undefined behavior triggered by the program |

# 2 | Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the NFT Staking implementation. 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 | No of Findings |
|---|---|
| Critical | |
| High | |
| Medium | |
| Low | |
| Informational | |
| Suggestions | |
| Total | |

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, reproduc- tion, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that 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.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues.

- 1 critical-severity vulnerability
- 2 medium-severity vulnerabilities
- 2 low-severity vulnerabilities.

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| 001 | Low | 3.1 Gas optimization | Patching | |
| 002 | Medium | 3.2 :Define a struct | Patching | |
| 003 | Medium | 3.3 :Remove extra Mapping | Patching | |
| 004 | High | 3.4 : Wrong Calculation | Data Validation | |
| 005 | High | 3.5 : Wrong Functionality | Denial of Service | |
| 006 | High | 3.6 : Multiple Findings In Stake Function. | Denial of Service | |
| 007 | High | 3.7: Multiple Findings in Withdraw Function. | Denial of Service | |
| 008 | Medium | 3.8: Remove extra functionality | Patching | |
| 009 | Low | 3.9: Change according to Functionality | Patching | |
| 010 | High | 3.10: Multiple Findings in Unstake Function | Denial of Service | |

| | | | | |
|---|---|---|---|---|
| 011 | High | 3.11: Wrong modifier | Denial of Service | |
| 012 | Low | 3.12: Extra Modifier | Data Validation | |
| 013 | High | 4.1: _Add a require check | Data Validation | |
| 014 | High | 4.2: _Apply check effect Interaction Pattern | Data Validation | |
| | | | | |

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 Gas optimization

### Description:

**line 1032:** No need to assign a zero value to the reward variable because it by default zero.In general, defining a simple variable, such as an integer, can cost around 200 gas, while assigning a value to it can cost around an additional 200 gas.

Line1033: The lastUpdateTime isnt need in the contract because it seems like an extra variable in the contract which isn't performing any specific function and costing extra gas fee.

```
1032    uint256 public rewardRate = 0;
1033    uint256 lastUpdateTime;
```

## 3.2 :Define a struct

### Description:

Both mapping rewards and noOfStakeTokens are costings more gas and seems to be extra code effort , Need to define a struct and put those values init and all the more required and then use all those values with a single mapping.Which will make contract more simple, save more gas cost and increase the code readability too. Remove the total stacked variable as there is no use of it in the contract and if in any case you have to show it on the front side then store it offChain to save any extra spending gas cost , it is an state variable which used a lot of gas cost because it is stored in the storage.

```
mapping(address => uint256) public rewards;

uint256 public totalStaked;
mapping(address => uint256) public noOfStakedTokens;
```

## 3.3 :Remove extra Mapping

### Description:

No need for this stakeAssetOwner variable to store the NFt owner here in this contract because you can get it through the Nft contract ownerOf method.

```
mapping(uint256 => address) public stakedAssetOwner;

```

## 3.4 : Wrong Calculation

### Description:

The method “earned(address account)” calculating the reward of the staking of the NFT and the formula using to calculate the reward doesn't seems to be right it multiply the tokens with timestamps and the no of stake Nfts and Then all the existing reward he had got first, it seems like it will drain the whole owner wallet and it's totally wrong.

```
function earned(address account) public view returns (uint256) {
    return
        (noOfStakedTokens[account] *
            (block.timestamp - lastUpdateTime) * rewardRate) + rewards[msg.sender];
}

```

## 3.5 : Wrong Functionality

### Description:

The method “totalReward ()” seems to be giving the total reward get by the user but actually it isn't , it is giving the total balance of the reward token instead of the reward he get while staking their NFts instead. You need a separate variable for storing that reward every time he gets that reward and do it separately for each user.

```
function totalReward() external view returns (uint256) {
    return rewardsToken.balanceOf(address(this));
}
```

## 3.6 : Multiple Findings In Stake Function.

### Description:

- There is no need to place the nonReentrant modifier in the stake function.
- The “updateReward(msg.sender)” is an extra modifier in the “stake()” function to remove it.
- Need to use a single Nft “Id” to stake Nft instead of an array of Nfts, change the array into a single variable and place that info how much nfts are staked by a user against every user in case.
- There is no check that who can stake Nfts , put a required statement that only Nfts owners can stake their nfts instead everyone calls this function.

- Remove the required statement checking the array length; there will be no need of that anymore.

- Remove the for loop used in the “stake()” method there will be no need for it any more. We should avoid loops for dynamic arrays in write functions. It can affect our contracts.**Don't write loops that are unbounded as this can hit the gas limit, causing your transaction to fail**.

- Remove the amount variable too from the “stake()” method too; there will be no need for it anymore.

- Remove the stakeAssetOwner mapping storing the Owner of each nft owner, there is no need of that it already exists in the nft contract we can get it from here it's an extra line of code which will cost extra gas fee.

- Put a required statement that users do not stake a single nft again and again.

- There is no time period for how long the user can stake or is staking its nfts and get reward for that.
- Remove the updateReward modifier it isn't doing anything.

```
function stake(uint256[] memory tokenIds) external nonReentrant whenNotPaused updateReward(msg.sender) {
    require(tokenIds.length != 0, "Staking: No tokenIds provided");

    uint256 amount;
    for (uint256 i = 0; i < tokenIds.length; i += 1) {
        // Transfer user's NFTs to the staking contract
        stakingToken.safeTransferFrom(msg.sender, address(this), tokenIds[i]);
        // Increment the amount which will be staked
        amount += 1;
        // Save who is the staker/depositor of the token
        stakedAssetOwner[tokenIds[i]] = msg.sender;
    }
    _stake(amount);
    emit Staked(msg.sender, amount, tokenIds);
}
```

## 3.7: Multiple Findings in Withdraw Function.

### Description:

- There is no need to place the nonReentrant modifier in the stake function.

- The “updateReward(msg.sender)” is an extra modifier in the “withdraw” function to remove it.

- Need to use a single Nft “Id” to withdraw Nft instead of an array of Nfts, change the array into a single variable.

- The required statement that only Nfts owners can withdraw their nfts uses the extra mapping we can get the owner of nft through nft contract .

- Remove the required statement checking the array length; there will be no need of that anymore.

- Remove the for loop used in the “withdraw()” method there will be no need for it any more. We should avoid loops for dynamic arrays in write functions. It can affect our contracts.**Don't write loops that are unbounded as this can hit the gas limit, causing your transaction to fail**.

- Remove the stakeAssetOwner mapping storing the address zero , there is no need for that ,it's an extra line of code which will cost extra gas fee.
- Remove the amount variable too from the “withdraw()” method too; there will be no need for it anymore.

- There should be a required statement that would check that the user cannot withdraw the already withdrawn stake.

```
function withdraw(uint256[] memory tokenIds) internal nonReentrant updateReward(msg.sender) {
    require(tokenIds.length != 0, "Staking: No tokenIds provided");

    uint256 amount;
    for (uint256 i = 0; i < tokenIds.length; i += 1) {
        // Check if the user who withdraws is the owner
        require(
            stakedAssetOwner[tokenIds[i]] == msg.sender,
            "Staking: Not the staker of the token or the token is not staked"
        );
        // Transfer NFTs back to the owner
        stakingToken.safeTransferFrom(address(this), msg.sender, tokenIds[i]);
        // Increment the amount which will be withdrawn
        amount += 1;
        // Cleanup stakedAssetOwner for the current tokenId
        stakedAssetOwner[tokenIds[i]] = address(0);
    }
    _withdraw(amount);

    emit Withdrawn(msg.sender, amount, tokenIds);
}
```

## 3.8: Remove extra functionality

### Description:

This is an extra function which is not required to put in the contract and it isn't doing the work properly. It should have it just telling the balance of the reward token instead of the total reward received by the user.So remove it.

```
function walletRewardCheck(address account) external view returns (uint256){
    return rewardsToken.balanceOf(account);
}
```

## 3.9: Change according to Functionality

### Description:

Change this GetReward function according to the struct you are making instead of those mappings.

```

function getReward() internal nonReentrant updateReward(msg.sender) {
    uint256 reward = rewards[msg.sender];
    if (reward > 0) {
        rewards[msg.sender] = 0;
        rewardsToken.safeTransfer(msg.sender, reward);
        emit RewardPaid(msg.sender, reward);
    }
}
```

## 3.10: Multiple Findings in Unstake Function

### Description:

- There must be a single id instead of an array in “unstake()” method.
- rewards[msg.sender] = earned(msg.sender); needs to be changed according to the new struct that will be defined instead of that mapping.
- withdraw(tokenIds);will get that id instead of the whole array.

```
    function unstake(uint256[] memory tokenIds) external {
        require(tokenIds.length != 0, "input tokenId cannot be 0");
        rewards[msg.sender] = earned(msg.sender);
        withdraw(tokenIds);
        getReward();
    }
```

## 3.11: Wrong modifier

### Description:

This modifier isn't right, it isn't doing anything according to contract logic, just a complex logic which isn't doing anything.

```
modifier updateReward(address account) {
    lastUpdateTime = block.timestamp;
    if (account != address(0)) {
        rewards[account] = earned(account);
    }
    _;
}
```

## 3.12: Extra Modifier

### Description:

Remove the and ReentrancyGuard it isn't going to be used.

```

ntract UpdatedERC721Staking is ERC721Holder, ReentrancyGuard, Ownable, Pausable {
  using SafeERC20 for IERC20;

```

# 4 | Findings After First Changes

**Contract address after First changing:**
https://goerli.etherscan.io/address/0x9b5798dff0acbe6176f435be71850053eebb1c91#code

## Now the new changing in new contract are:

### 4.1:_ Add a require check

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Target: StakingContract
- Category: Business Logic [4]
- CWE subcategory: CWE

### Description:

There is no check on the token length that wouldn't be zero in unstake() method.

# Now the new changing in new contract are:

## 4.2:_ Apply check effect Interaction Pattern

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Target: StakingContract
- Category: Business Logic [4]
- CWE subcategory: CWE

### Description:

Apply a check effect pattern on staking contracts that there will be no chance of minor attacks and security risks.