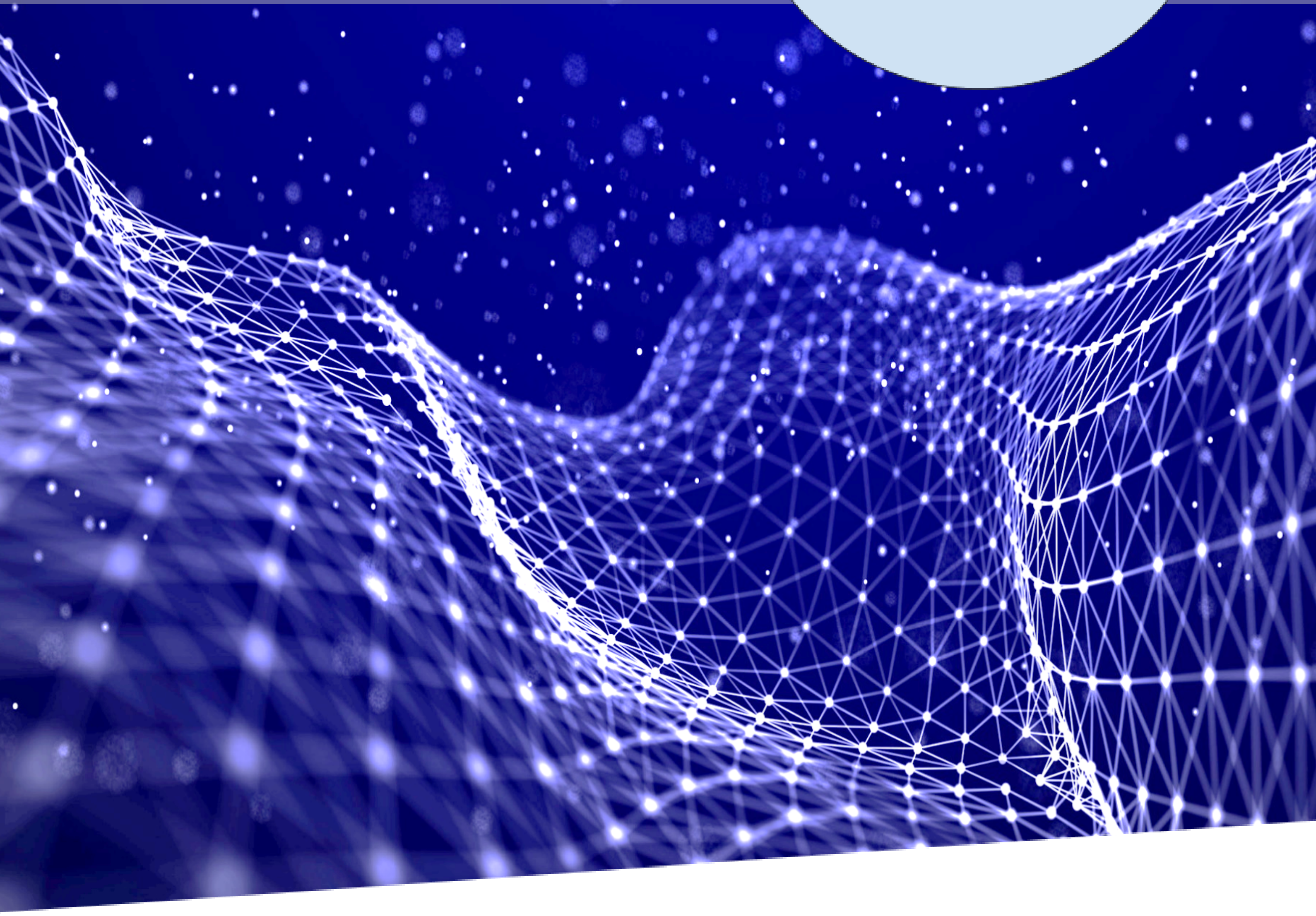
# PlayToEarn NFTMarketplace

## Smart Contract Audit Report



**Date Issued:** 11 Jan 2022

**Version:** Final v1.0

*Public*

# Table of Contents

# Executive Summary

## Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **NFTMarketplace feature**. This audit report was published on *11 Jan 2022*. The audit scope is limited to the **NFTMarketplace feature.** Our security best practices strongly recommend that the **PlayToEarn team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

## About NFTMarketplace

**PlayToEarn Marketplace** is where gamers can easily trade NFTs from beloved games and support the game developers. Users can easily search through the entire marketplace to find what they want. The platform provides a detailed description of each NFT as well as market conditions for the most informed decision. These NFTs can be redeemed in our partners' ecosystem to put those items in use. Game creators can earn market fees from listing their game NFTs on our marketplace with high customization capabilities to provide the most smooth experience for gamers.

## Scope of Work

The security audit conducted does not replace the full security audit of the overall PlayToEarn protocol. The scope is limited to the **NFTMarketplace feature** and their related smart contracts.

The security audit covered the components at this specific state:

| Item | Description |
|---|---|
| Components | ▪ *NFTMarketplace smart contract*<br>▪ *Imported associated smart contracts and libraries* |
| GitHub Repository | ▪ *https://github.com/playtoearndev/playtoearn-nft-marketplace-contracts* |
| Commit | ▪ *c5f93fbfa90791e4772f74f2cb423735f914c098* |
| Reassessment Commit | ▪ *871b45d44067c84a4495af2cdc1dc313c975c48e* |
| Audited Files | ▪ *contracts/NFTMarketplace.sol* |

| | |
|---|---|
| **Excluded Files/Contracts** | - |

*Remark: Our security best practices strongly recommend that the PlayToEarn team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.*

# Auditors

Phuwanai Thummavet

Sumedt Jitpukdebodin

# Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

# Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **NFTMarketplace feature** has sufficient security protections to be safe for use.

Verified by Valix Consulting

The smart contracts PASS our security verification and qualification level.

Verified on 11 Jan 2022

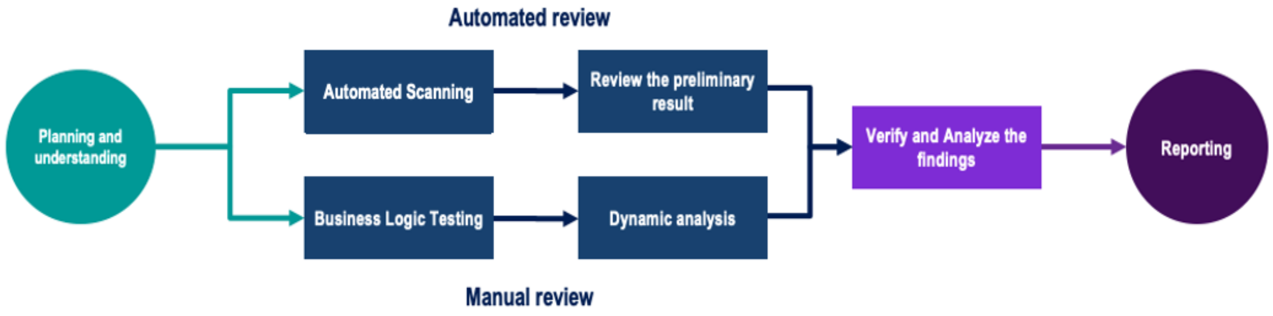
Initially, Valix was able to identify **18 issues** that were categorized from the “Critical” to “Informational” risk level in the given timeframe of the assessment. For the reassessment, the PlayToEarn team fully fixed 17 issues and partially fixed 1 medium issue. Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

| Target | Assessment Result | | | | | Reassessment Result | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | C | H | M | L | I | C | H | M | L | I |
| **NFTMarketplace** | 1 | 3 | 7 | 3 | 4 | 0 | 0 | 1 | 0 | 0 |

***Note:*** *Risk Rating* **C** *Critical,* **H** *High,* **M** *Medium,* **L** *Low,* **I** *Informational*

# Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



### Planning and Understanding

- Determine the scope of testing and understanding the application’s purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

### Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

### Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

### Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

# Audit Items

We perform the audit according to the following categories and test names.

| Category | ID | Test Name |
|---|---|---|
| Security Issue | SEC01 | *Authorization Through tx.origin* |
| | SEC02 | *Business Logic Flaw* |
| | SEC03 | *Delegatecall to Untrusted Callee* |
| | SEC04 | *DoS With Block Gas Limit* |
| | SEC05 | *DoS with Failed Call* |
| | SEC06 | *Function Default Visibility* |
| | SEC07 | *Hash Collisions With Multiple Variable Length Arguments* |
| | SEC08 | *Incorrect Constructor Name* |
| | SEC09 | *Improper Access Control or Authorization* |
| | SEC10 | *Improper Emergency Response Mechanism* |
| | SEC11 | *Insufficient Validation of Address Length* |
| | SEC12 | *Integer Overflow and Underflow* |
| | SEC13 | *Outdated Compiler Version* |
| | SEC14 | *Outdated Library Version* |
| | SEC15 | *Private Data On-Chain* |
| | SEC16 | *Reentrancy* |
| | SEC17 | *Transaction Order Dependence* |
| | SEC18 | *Unchecked Call Return Value* |
| | SEC19 | *Unexpected Token Balance* |
| | SEC20 | *Unprotected Assignment of Ownership* |
| | SEC21 | *Unprotected SELFDESTRUCT Instruction* |
| | SEC22 | *Unprotected Token Withdrawal* |
| | SEC23 | *Unsafe Type Inference* |
| | SEC24 | *Use of Deprecated Solidity Functions* |
| | SEC25 | *Use of Untrusted Code or Libraries* |
| | SEC26 | *Weak Sources of Randomness from Chain Attributes* |
| | SEC27 | *Write to Arbitrary Storage Location* |

| Category | ID | Test Name |
|---|---|---|
| **Functional Issue** | **FNC01** | *Arithmetic Precision* |
| | **FNC02** | *Permanently Locked Fund* |
| | **FNC03** | *Redundant Fallback Function* |
| | **FNC04** | *Timestamp Dependence* |
| **Operational Issue** | **OPT01** | *Code With No Effects* |
| | **OPT02** | *Message Call with Hardcoded Gas Amount* |
| | **OPT03** | *The Implementation Contract Flow or Value and the Document is Mismatched* |
| | **OPT04** | *The Usage of Excessive Byte Array* |
| | **OPT05** | *Unenforced Timelock on An Upgradeable Proxy Contract* |
| **Developmental Issue** | **DEV01** | *Assert Violation* |
| | **DEV02** | *Other Compilation Warnings* |
| | **DEV03** | *Presence of Unused Variables* |
| | **DEV04** | *Shadowing State Variables* |
| | **DEV05** | *State Variable Default Visibility* |
| | **DEV06** | *Typographical Error* |
| | **DEV07** | *Uninitialized Storage Pointer* |
| | **DEV08** | *Violation of Solidity Coding Convention* |
| | **DEV09** | *Violation of Token (ERC20) Standard API* |

# Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

| Risk Level | Definition |
|---|---|
| Critical | The code implementation does not match the specification, and it could disrupt the platform. |
| High | The code implementation does not match the specification, or it could result in the loss of funds for contract owners or users. |
| Medium | The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control. |
| Low | The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line. |
| Informational | Findings in this category are informational and may be further improved by following best practices and guidelines. |

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

| Likelihood<br>Impact | High | Medium | Low |
|---|---|---|---|
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Informational |

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

# Findings

## Review Findings Summary

The table below shows the summary of our assessments.

| No. | Issue | Risk | Status | Functionality is in use |
|---|---|---|---|---|
| 1 | Potential Theft Of All NFT Assets | Critical | Fixed | In use |
| 2 | Selling NFT Assets Without Updating Remaining Amount | High | Fixed | In use |
| 3 | Contract Parameters Can Be Adjusted Without Time Delay | High | Fixed | In use |
| 4 | Existence Of Risky Function | High | Fixed | In use |
| 5 | Invalid Struct Design | Medium | Fixed | In use |
| 6 | Possible Denial Of Service On NFT Data Querying | Medium | Fixed | In use |
| 7 | Unsafe Function Use | Medium | Fixed | In use |
| 8 | Setting Fee Without Limit | Medium | Fixed | In use |
| 9 | Possibly Permanent Ownership Removal | Medium | Fixed | In use |
| 10 | Unsafe Ownership Transfer | Medium | Fixed | In use |
| 11 | Improper NFT Data Querying | Medium | Partially Fixed | In use |
| 12 | No Input Sanitization Checks | Low | Fixed | In use |
| 13 | The Compiler Is Not Locked To A Specific Version | Low | Fixed | In use |
| 14 | The Compiler May Be Susceptible To The Publicly Disclosed Bugs | Low | Fixed | In use |
| 15 | Recommended Gas Optimization | Informational | Fixed | In use |
| 16 | Misleading Struct Field | Informational | Fixed | In use |
| 17 | Misleading State Variable | Informational | Fixed | In use |
| 18 | Inconsistent Comment With The Code | Informational | Fixed | In use |

The statuses of the issues are defined as follows:

**Fixed:** The issue has been completely resolved and has no further complications.

**Partially Fixed:** The issue has been partially resolved.

**Acknowledged:** The issue's risk has been reported and acknowledged.

# Detailed Result

This section provides all issues that we found in detail.

| No. 1 | Potential Theft Of All NFT Assets | | |
|---|---|---|---|
| Risk | Critical | Likelihood | High |
| | | Impact | High |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 161 - 214 and 216 - 250* | | |

## Detailed Issue

We found the critical vulnerability on the ***buyMarketItem*** (L161 - 214) and ***unlistMarketItem*** (L216 - 250) functions that enable an attacker to steal all NFT assets on the marketplace. For brevity's sake, we will explain this issue by describing the case of the *buyMarketItem* function since the vulnerability on the *unlistMarketItem* function is almost identical.

Consider the following attack scenario.

1. Bob places his NFT assets for trading by executing the function *createMarketItem(nftContract: **0xabc..012**, tokenId: **1**, price: **500**, amount: **10**)*. As an execution result, the function generates a sequent id ***itemId: 1*** for Bob's NFT assets.

2. The attacker sees Bob's NFT assets on the marketplace. The attacker then creates the forged NFT assets by calling the function *createMarketItem(nftContract: **0xdef..666**, tokenId: **1**, price: **1**, amount: **10**)*. Note that ***0xdef..666*** is the NFT contract address imitated for stealing Bob's NFT assets created in Step 1. The function generates the next id ***itemId: 2*** for the attacker's NFT assets as an execution result.

3. The attacker manages to steal Bob's NFT assets by invoking the function *buyMarketItem(nftContract: **0xabc..012**, itemId: **2**, amount: **10**)*. Consequently, the attacker can steal Bob's NFT assets (***itemId: 1***) by manipulating the *nftContract* parameter.

The root cause of this issue is that the attacker can manipulate the *nftContract* parameter (L162 in the code snippet below) by specifying an address of Bob's NFT contract (***0xabc..012***) whereas specifying the forged NFT assets (***itemId: 2***) to bypass the computation from L166 to L182.

Eventually, the *buyMarketItem* function will transfer the managed NFT assets from the *NFTMarketplace* contract to the attacker address (L184 - 190).

**NFTMarketplace.sol**

```
function buyMarketItem(
    address nftContract,
    uint256 itemId,
    uint256 amount
) public nonReentrant {
    uint256 price = idToMarketItem[itemId].price;
    uint256 tokenId = idToMarketItem[itemId].tokenId;
    uint256 fee = calculateFee(amount, price);

    require(amount > 0, "Amount must > 0");
    require(
        idToMarketItem[itemId].amount >= amount,
        "Insufficient market item amount"
    );
    require(idToMarketItem[itemId].isSold != true, "This item is sold");
    require(idToMarketItem[itemId].isUnlisted != true, "This item is unlisted");

    uint256 cost = idToMarketItem[itemId].price.mul(amount).sub(fee);
    require(_currency.balanceOf(msg.sender) >= cost, "Insufficient currency");

    // Transfer currency to contract owner
    _currency.transferFrom(msg.sender, idToMarketItem[itemId].seller, cost);

    IERC1155(nftContract).safeTransferFrom(
        address(this),
        msg.sender,
        tokenId,
        amount,
        "0x0"
    );

    idToMarketItem[itemId].owner = msg.sender;

    // Transfer fee to contract owner
    _currency.transferFrom(msg.sender, owner(), fee);

    bool sold = idToMarketItem[itemId].amount == amount;
    if (sold) {
        idToMarketItem[itemId].isSold = true;
        _itemsSold.increment();
    }

    emit MarketItemSold(
        itemId,
        nftContract,
        idToMarketItem[itemId].tokenId,
        idToMarketItem[itemId].seller,
        idToMarketItem[itemId].owner,
        idToMarketItem[itemId].price,
210             amount,
211             sold,
212             false
213         );
214     }
```

Listing 1.1 The *buyMarketItem* function that is vulnerable to NFT theft

## Recommendations

We recommend updating both the ***buyMarketItem*** (L161 - 214) and ***unlistMarketItem*** (L216 - 250) functions by employing the ***idToMarketItem[itemId].nftContract*** instead of the manipulatable parameter ***nftContract*** (L183) and removing the parameter ***nftContract*** from the functions like the code snippet below.

**NFTMarketplace.sol**

```
161 function buyMarketItem(
162     uint256 itemId,
163     uint256 amount
164 ) public nonReentrant {
165     uint256 price = idToMarketItem[itemId].price;
166     uint256 tokenId = idToMarketItem[itemId].tokenId;
167     uint256 fee = calculateFee(amount, price);
168 
169     require(amount > 0, "Amount must > 0");
170     require(
171         idToMarketItem[itemId].amount >= amount,
172         "Insufficient market item amount"
173     );
174     require(idToMarketItem[itemId].isSold != true, "This item is sold");
175     require(idToMarketItem[itemId].isUnlisted != true, "This item is unlisted");
176 
177     uint256 cost = idToMarketItem[itemId].price.mul(amount).sub(fee);
178     require(_currency.balanceOf(msg.sender) >= cost, "Insufficient currency");
179 
180     // Transfer currency to contract owner
181     _currency.transferFrom(msg.sender, idToMarketItem[itemId].seller, cost);
182 
183     IERC1155(idToMarketItem[itemId].nftContract).safeTransferFrom(
184         address(this),
185         msg.sender,
186         tokenId,
187         amount,
188         "0x0"
189     );
190 
191     idToMarketItem[itemId].owner = msg.sender;
192 
```

```
    // Transfer fee to contract owner
    _currency.transferFrom(msg.sender, owner(), fee);

    bool sold = idToMarketItem[itemId].amount == amount;
    if (sold) {
        idToMarketItem[itemId].isSold = true;
        _itemsSold.increment();
    }

    emit MarketItemSold(
        itemId,
        nftContract,
        idToMarketItem[itemId].tokenId,
        idToMarketItem[itemId].seller,
        idToMarketItem[itemId].owner,
        idToMarketItem[itemId].price,
        amount,
        sold,
        false
    );
}
```

Listing 1.2 The improved *buyMarketItem* function

## Reassessment

The PlayToEarn team fixed this issue by employing the ***idToMarketItem[itemId].nftContract*** instead of the manipulatable parameter ***nftContract*** and removing the parameter ***nftContract*** from the associated functions according to our recommendation.

| No. 2 | Selling NFT Assets Without Updating Remaining Amount | | |
|---|---|---|---|
| Risk | High | Likelihood | High |
| | | Impact | Medium |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 161 - 214* | | |

## Detailed Issue

As shown in the code snippet below, we found that the *`buyMarketItem`* function sells NFT assets without updating the remaining amount. This inconsistent record may lead to unexpected errors or even denial-of-service issues to the platform.

**NFTMarketplace.sol**

```
161 function buyMarketItem(
162     address nftContract,
163     uint256 itemId,
164     uint256 amount
165 ) public nonReentrant {
166     uint256 price = idToMarketItem[itemId].price;
167     uint256 tokenId = idToMarketItem[itemId].tokenId;
168     uint256 fee = calculateFee(amount, price);
169 
170     require(amount > 0, "Amount must > 0");
171     require(
172         idToMarketItem[itemId].amount >= amount,
173         "Insufficient market item amount"
174     );
175     require(idToMarketItem[itemId].isSold != true, "This item is sold");
176     require(idToMarketItem[itemId].isUnlisted != true, "This item is unlisted");
177 
178     uint256 cost = idToMarketItem[itemId].price.mul(amount).sub(fee);
179     require(_currency.balanceOf(msg.sender) >= cost, "Insufficient currency");
180 
181     // Transfer currency to contract owner
182     _currency.transferFrom(msg.sender, idToMarketItem[itemId].seller, cost);
183 
184     IERC1155(nftContract).safeTransferFrom(
185         address(this),
186         msg.sender,
187         tokenId,
188         amount,
189         "0x0"
190     );
191 
192     idToMarketItem[itemId].owner = msg.sender;
193 
194     // Transfer fee to contract owner
195     _currency.transferFrom(msg.sender, owner(), fee);
196 
197     bool sold = idToMarketItem[itemId].amount == amount;
198     if (sold) {
199         idToMarketItem[itemId].isSold = true;
200         _itemsSold.increment();
201     }
202 
203     emit MarketItemSold(
204         itemId,
205         nftContract,
206         idToMarketItem[itemId].tokenId,
207         idToMarketItem[itemId].seller,
208         idToMarketItem[itemId].owner,
209         idToMarketItem[itemId].price,
210         amount,
211         sold,
212         false
213     );
214 }
```

Listing 2.1 The *buyMarketItem* function sells NFT assets without updating the remaining amount

## Recommendations

We recommend updating the remaining amount after NFT assets are purchased, like the example code snippet below (L197 - 198).

**NFTMarketplace.sol**

```
161 function buyMarketItem(
162     address nftContract,
163     uint256 itemId,
164     uint256 amount
165 ) public nonReentrant {
166     uint256 price = idToMarketItem[itemId].price;
167     uint256 tokenId = idToMarketItem[itemId].tokenId;
168     uint256 fee = calculateFee(amount, price);
169 
170     require(amount > 0, "Amount must > 0");
171     require(
172       idToMarketItem[itemId].amount >= amount,
      "Insufficient market item amount"
    );
    require(idToMarketItem[itemId].isSold != true, "This item is sold");
    require(idToMarketItem[itemId].isUnlisted != true, "This item is unlisted");

    uint256 cost = idToMarketItem[itemId].price.mul(amount).sub(fee);
    require(_currency.balanceOf(msg.sender) >= cost, "Insufficient currency");

    // Transfer currency to contract owner
    _currency.transferFrom(msg.sender, idToMarketItem[itemId].seller, cost);

    IERC1155(nftContract).safeTransferFrom(
      address(this),
      msg.sender,
      tokenId,
      amount,
      "0x0"
    );

    idToMarketItem[itemId].owner = msg.sender;

    // Transfer fee to contract owner
    _currency.transferFrom(msg.sender, owner(), fee);

    idToMarketItem[itemId].amount = idToMarketItem[itemId].amount.sub(amount);
    if (idToMarketItem[itemId].amount == 0) {
      idToMarketItem[itemId].isSold = true;
      _itemsSold.increment();
    }

    emit MarketItemSold(
      itemId,
      nftContract,
      idToMarketItem[itemId].tokenId,
      idToMarketItem[itemId].seller,
      idToMarketItem[itemId].owner,
      idToMarketItem[itemId].price,
      amount,
      sold,
      false
    );
}
```

Listing 2.2 The improved *buyMarketItem* function

## Reassessment

The PlayToEarn team fixed this issue according to our recommendation.

| **No. 3** | **Contract Parameters Can Be Adjusted Without Time Delay** | | |
|---|---|---|---|
| **Risk** | **High** | **Likelihood** | **Medium** |
| | | **Impact** | **High** |
| **Functionality is in use** | **In use** | **Status** | **Fixed** |
| **Associated Files** | *contracts/NFTMarketplace.sol* | | |
| **Locations** | *NFTMarketplace.sol L: 103 - 106 and 112 - 115* | | |

## Detailed Issue

The developer can adjust the contract parameters immediately, affecting the platform's trustworthiness and raising concerns to users.

The code snippet below shows the *`setFee`* and *`setCurrency`* functions that allow the developer to adjust the *`fee`* and *`currency token`* freely. We consider that changing the *`fee`* or *`currency token`* can affect the value of users' NFT assets in the marketplace.

**NFTMarketplace.sol**

```
103  function setFee(uint256 fee) public onlyOwner {
104      _fee = fee;
105      emit SetFee(fee);
106  }

     (...SNIP...)

112  function setCurrency(address currency) public onlyOwner {
113      _currency = IERC20(currency);
114      emit SetCurrency(currency);
115  }
```

Listing 3.1 The *`setFee`* and *`setCurrency`* functions enable the developer to adjust the *`fee`* and *`currency token`* freely

## Recommendations

We recommend applying the *Timelock* contract to the *NFTMarketplace* contract. The relationship between each entity should be as follows:

***Developer address -> Timelock -> NFTMarketplace***

Every time a developer adjusts any contract parameters, the *Timelock* will defer the transaction for some waiting period (e.g., 48 hours) configured. This enables users to examine what parameters the developer wants to adjust before effective, providing transparency.

## Reassessment

The PlayToEarn team would fix this issue by employing the *OpenZeppelin Defender* to deploy the *Timelock* for the *NFTMarketplace* contract.

| No. 4 | Existence Of Risky Function | | |
|---|---|---|---|
| Risk | High | Likelihood | Medium |
| | | Impact | High |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 112 - 115* | | |

## Detailed Issue

The developer can change a *currency token* used as the medium of exchange for NFT trading through the *setCurrency* function (L112), as shown in the code snippet below.

**NFTMarketplace.sol**

```
112 function setCurrency(address currency) public onlyOwner {
113     _currency = IERC20(currency);
114     emit SetCurrency(currency);
115 }
```

Listing 4.1 The *setCurrency* function allows the developer to change the *currency token*

We found that the *change of currency token* via the *setCurrency* function can affect the value of users' NFT assets on the marketplace. Let's consider the following scenario to understand this issue.

1. The developer deploys the *NFTMarketplace* contract and sets the *BNB token* (via a *contract constructor*) as the medium of exchange.
2. Bob places his NFT asset and sets its price at *1 BNB* (assuming that *1 BNB* equals *$600*) for sale on the marketplace.
3. The developer changes the currency token from *BNB* to *USDT* token via the *setCurrency* function.
4. Bob's NFT asset value is lowered from *$600* to *$1* immediately.
5. Alice sells Bob's NFT with *1 USDT*.

We consider changing the *currency token* while there are NFT assets open for sale on the marketplace dangerous. In other words, the marketplace should use the fixed currency token.

## Recommendations

We consider the *`setCurrency`* function risky for the platform and recommend removing it from the *NFTMarketplace* contract. The contract should use the fixed currency token.

## Reassessment

According to our recommendation, the PlayToEarn team fixed this issue by removing the *`setCurrency`* function from the *NFTMarketplace* contract.

| No. 5 | Invalid Struct Design | | |
|---|---|---|---|
| Risk | Medium | Likelihood | High |
| | | Impact | Low |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 32 - 42, 161 - 214, 287 - 302 and 312 - 333* | | |

## Detailed Issue

Since the *NFTMarketplace* contract supports multi-asset trading on the same item id, multiple users can own assets on the same item id. However, our investigation found that the *MarketItem* struct (L32 - 42 in the code snippet 5.1) used to track NFT assets of each token id has an invalid design.

The *MarketItem* struct supports only a single owner tracking at a time (L37 in the code snippet 5.1). Let's consider the *buyMarketItem* function (L192 in the code snippet 5.2). The *buyMarketItem* function will overwrite the NFT owner every time the remaining assets are purchased.

This invalid struct design may lead to incorrect querying results of the following functions.

1. *getMarketItems* function (L287 - 302)

2. *fetchPurchasedNFTs* function (L312 - 333)

The code snippet 5.3 shows one of the affected functions, *fetchPurchasedNFTs*, that may return incorrect querying results because of the invalid struct design.

**NFTMarketplace.sol**

```
32  struct MarketItem {
33      uint256 itemId;
34      address nftContract;
35      uint256 tokenId;
36      address seller;
37      address owner;
38      uint256 price;
39      uint256 amount;
40      bool isSold;
41      bool isUnlisted;
42  }
```

Listing 5.1 The *MarketItem* struct supporting only a single owner

**NFTMarketplace.sol**

```
function buyMarketItem(
    address nftContract,
    uint256 itemId,
    uint256 amount
) public nonReentrant {
    uint256 price = idToMarketItem[itemId].price;
    uint256 tokenId = idToMarketItem[itemId].tokenId;
    uint256 fee = calculateFee(amount, price);

    require(amount > 0, "Amount must > 0");
    require(
        idToMarketItem[itemId].amount >= amount,
        "Insufficient market item amount"
    );
    require(idToMarketItem[itemId].isSold != true, "This item is sold");
    require(idToMarketItem[itemId].isUnlisted != true, "This item is unlisted");

    uint256 cost = idToMarketItem[itemId].price.mul(amount).sub(fee);
    require(_currency.balanceOf(msg.sender) >= cost, "Insufficient currency");

    // Transfer currency to contract owner
    _currency.transferFrom(msg.sender, idToMarketItem[itemId].seller, cost);

    IERC1155(nftContract).safeTransferFrom(
        address(this),
        msg.sender,
        tokenId,
        amount,
        "0x0"
    );

    idToMarketItem[itemId].owner = msg.sender;

    // Transfer fee to contract owner
    _currency.transferFrom(msg.sender, owner(), fee);

    bool sold = idToMarketItem[itemId].amount == amount;
    if (sold) {
        idToMarketItem[itemId].isSold = true;
        _itemsSold.increment();
    }

    emit MarketItemSold(
        itemId,
        nftContract,
        idToMarketItem[itemId].tokenId,
        idToMarketItem[itemId].seller,
        idToMarketItem[itemId].owner,
        idToMarketItem[itemId].price,
        amount,
        sold,
        false
    );
}
```

Listing 5.2 The *buyMarketItem* function will overwrite the NFT owner every time the remaining assets are purchased

**NFTMarketplace.sol**

```
function fetchPurchasedNFTs() public view returns (MarketItem[] memory) {
    uint256 totalItemCount = _itemIds.current();
    uint256 itemCount = 0;
    uint256 currentIndex = 0;

    for (uint256 i = 0; i < totalItemCount; i++) {
      if (idToMarketItem[i + 1].owner == msg.sender) {
        itemCount += 1;
      }
    }

    MarketItem[] memory marketItems = new MarketItem[](itemCount);
    for (uint256 i = 0; i < totalItemCount; i++) {
      if (idToMarketItem[i + 1].owner == msg.sender) {
        uint256 currentId = idToMarketItem[i + 1].itemId;
        MarketItem storage currentItem = idToMarketItem[currentId];
        marketItems[currentIndex] = currentItem;
        currentIndex += 1;
      }
    }
    return marketItems;
}
```

Listing 5.3 The *fetchPurchasedNFTs* function may return incorrect querying results

## Recommendations

We recommend re-designing/implementing the *`MarketItem`* struct to support multiple owners tracking. The code snippet below shows an example solution to multi-owner tracking.

**NFTMarketplace.sol**

```
struct OwnerInfo {
    address owner;
    uint256 amount;
}

struct MarketItem {
    uint256 itemId;
    address nftContract;
    uint256 tokenId;
    address seller;
    OwnerInfo[] ownerInfo;
    uint256 price;
    uint256 amount;
    bool isSold;
    bool isUnlisted;
}
```

Listing 5.4 The *`MarketItem`* struct that supports multiple owners tracking

## Reassessment

The PlayToEarn team fixed this issue by tracking multiple owners under the same item id as the code snippet below.

**NFTMarketplace.sol**

```
struct OwnerInfo {
    address owner;
    uint256 amount;
    uint256 atBlock;
}

struct MarketItem {
    uint256 itemId;
    address nftContract;
    uint256 tokenId;
    address seller;
    mapping(uint256 => OwnerInfo) ownerInfo;
    Counters.Counter ownerInfoCount;
    uint256 price;
    uint256 amount;
    bool isSold;
    bool isUnlisted;
}
```

Listing 5.5 The fixed `MarketItem` struct

| No. 6 | Possible Denial Of Service On NFT Data Querying | | |
|---|---|---|---|
| Risk | Medium | Likelihood | Low |
| | | Impact | High |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 287 - 302, 312 - 333, and 335 - 356* | | |

## Detailed Issue

On the NFTMarketplace platform, the number of NFT assets placed for trading might grow over time. With this assumption, we found that the following *`view`* functions might confront a denial-of-service issue if the number of NFT assets on the marketplace is too large.

The affected functions include:

1. *`getMarketItems`* function (L287 - 302)
2. *`fetchPurchasedNFTs`* function (L312 - 333)
3. *`fetchCreateNFTs`* function (L335 - 356)

The root cause of this issue is that the affected functions have to iterate over all NFT assets (L293, L317, L324, L340, and L347 in the code snippet below), which might take too long for querying on the EVM node, leading to the rejection of querying request.

NFTMarketplace.sol

```
function getMarketItems() public view returns (MarketItem[] memory) {
    uint256 itemCount = _itemIds.current();
    uint256 unsoldItemCount = _itemIds.current() - _itemsSold.current();
    uint256 currentIndex = 0;

    MarketItem[] memory marketItems = new MarketItem[](unsoldItemCount);
    for (uint256 i = 0; i < itemCount; i++) {
      if (idToMarketItem[i + 1].owner == address(0)) {
        uint256 currentId = idToMarketItem[i + 1].itemId;
        MarketItem storage currentItem = idToMarketItem[currentId];
        marketItems[currentIndex] = currentItem;
        currentIndex += 1;
      }
    }
    return marketItems;
}

(...SNIP...)

function fetchPurchasedNFTs() public view returns (MarketItem[] memory) {
    uint256 totalItemCount = _itemIds.current();
    uint256 itemCount = 0;
    uint256 currentIndex = 0;

    for (uint256 i = 0; i < totalItemCount; i++) {
      if (idToMarketItem[i + 1].owner == msg.sender) {
        itemCount += 1;
      }
    }

    MarketItem[] memory marketItems = new MarketItem[](itemCount);
    for (uint256 i = 0; i < totalItemCount; i++) {
      if (idToMarketItem[i + 1].owner == msg.sender) {
        uint256 currentId = idToMarketItem[i + 1].itemId;
        MarketItem storage currentItem = idToMarketItem[currentId];
        marketItems[currentIndex] = currentItem;
        currentIndex += 1;
      }
    }
    return marketItems;
}

function fetchCreateNFTs() public view returns (MarketItem[] memory) {
    uint256 totalItemCount = _itemIds.current();
    uint256 itemCount = 0;
    uint256 currentIndex = 0;

    for (uint256 i = 0; i < totalItemCount; i++) {
      if (idToMarketItem[i + 1].seller == msg.sender) {
        itemCount += 1; // No dynamic length. Predefined length has to be made
      }
    }

    MarketItem[] memory marketItems = new MarketItem[](itemCount);
    for (uint256 i = 0; i < totalItemCount; i++) {
      if (idToMarketItem[i + 1].seller == msg.sender) {
        uint256 currentId = idToMarketItem[i + 1].itemId;
        MarketItem storage currentItem = idToMarketItem[currentId];
        marketItems[currentIndex] = currentItem;
        currentIndex += 1;
      }
    }
    return marketItems;
}
```

Listing 6.1 The *getMarketItems*, *fetchPurchasedNFTs*, and *fetchCreateNFTs* functions that are prone to the denial-of-service issue

## Recommendations

We recommend two possible solutions. The first solution is to apply pagination for data querying, in which the large querying data are divided into smaller discrete pages.

The second solution is to employ different arrays for tracking different NFT assets of interest. For example, using different arrays to track assets available for sale, purchased assets, and assets created by a specific seller.

## Reassessment

According to our suggestion, the PlayToEarn team fixed this issue by applying pagination for data querying. Our further recommendation is to make query calls at the same block number for consistent querying results.

Unfortunately, we found a further issue with the improved functions during the reassessment of this issue. Please refer to issue no. 11 for more details.

| No. 7 | Unsafe Function Use | | |
|---|---|---|---|
| Risk | Medium | Likelihood | Low |
| | | Impact | High |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 182 and 195* | | |

## Detailed Issue

The *buyMarketItem* function uses an unsafe ERC-20 *transferFrom* function (L182 and L195 in the code snippet below) that can lead to unexpected ERC-20 transfer errors.

**NFTMarketplace.sol**

```
161  function buyMarketItem(
162      address nftContract,
163      uint256 itemId,
164      uint256 amount
165  ) public nonReentrant {
166      uint256 price = idToMarketItem[itemId].price;
167      uint256 tokenId = idToMarketItem[itemId].tokenId;
168      uint256 fee = calculateFee(amount, price);
169
170      require(amount > 0, "Amount must > 0");
171      require(
172        idToMarketItem[itemId].amount >= amount,
173        "Insufficient market item amount"
174      );
175      require(idToMarketItem[itemId].isSold != true, "This item is sold");
176      require(idToMarketItem[itemId].isUnlisted != true, "This item is unlisted");
177
178      uint256 cost = idToMarketItem[itemId].price.mul(amount).sub(fee);
179      require(_currency.balanceOf(msg.sender) >= cost, "Insufficient currency");
180
181      // Transfer currency to contract owner
182      _currency.transferFrom(msg.sender, idToMarketItem[itemId].seller, cost);
183
184      IERC1155(nftContract).safeTransferFrom(
185        address(this),
186        msg.sender,
187        tokenId,
188        amount,
189       "0x0"
190     );
191 
192     idToMarketItem[itemId].owner = msg.sender;
193 
194     // Transfer fee to contract owner
195     _currency.transferFrom(msg.sender, owner(), fee);
196 
197     bool sold = idToMarketItem[itemId].amount == amount;
198     if (sold) {
199       idToMarketItem[itemId].isSold = true;
200       _itemsSold.increment();
201     }
202 
203     emit MarketItemSold(
204       itemId,
205       nftContract,
206       idToMarketItem[itemId].tokenId,
207       idToMarketItem[itemId].seller,
208       idToMarketItem[itemId].owner,
209       idToMarketItem[itemId].price,
210       amount,
211       sold,
212       false
213     );
214 }
```

Listing 7.1 The *buyMarketItem* function uses an unsafe *transferFrom* function

## Recommendations

We recommend applying the *safeTransferFrom* function of the *SafeERC20* library instead for safe ERC-20 transfer, as shown in L182 and L195 in the code snippet below.

**NFTMarketplace.sol**

```
161 function buyMarketItem(
162     address nftContract,
163     uint256 itemId,
164     uint256 amount
165 ) public nonReentrant {
166     uint256 price = idToMarketItem[itemId].price;
167     uint256 tokenId = idToMarketItem[itemId].tokenId;
168     uint256 fee = calculateFee(amount, price);
169 
170     require(amount > 0, "Amount must > 0");
171     require(
172       idToMarketItem[itemId].amount >= amount,
173       "Insufficient market item amount"
```

```
    );
    require(idToMarketItem[itemId].isSold != true, "This item is sold");
    require(idToMarketItem[itemId].isUnlisted != true, "This item is unlisted");

    uint256 cost = idToMarketItem[itemId].price.mul(amount).sub(fee);
    require(_currency.balanceOf(msg.sender) >= cost, "Insufficient currency");

    // Transfer currency to contract owner
    _currency.safeTransferFrom(msg.sender, idToMarketItem[itemId].seller, cost);

    IERC1155(nftContract).safeTransferFrom(
      address(this),
      msg.sender,
      tokenId,
      amount,
      "0x0"
    );

    idToMarketItem[itemId].owner = msg.sender;

    // Transfer fee to contract owner
    _currency.safeTransferFrom(msg.sender, owner(), fee);

    bool sold = idToMarketItem[itemId].amount == amount;
    if (sold) {
      idToMarketItem[itemId].isSold = true;
      _itemsSold.increment();
    }

    emit MarketItemSold(
      itemId,
      nftContract,
      idToMarketItem[itemId].tokenId,
      idToMarketItem[itemId].seller,
      idToMarketItem[itemId].owner,
      idToMarketItem[itemId].price,
      amount,
      sold,
      false
    );
}
```

Listing 7.2 The improved `buyMarketItem` function that uses the `safeTransferFrom` function

## Reassessment

The PlayToEarn team fixed this issue as per our recommendation.

| No. 8 | Setting Fee Without Limit | | |
|---|---|---|---|
| Risk | Medium | Likelihood | Medium |
| | | Impact | Medium |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 27 - 30 and 103 - 106* | | |

## Detailed Issue

The change of *platform fee* affects the income of users directly. However, the developer can set the *platform fee* without limit through the *constructor* (L27 - 30) and *setFee* (L103 - 106) function, as shown in the code snippet below.

**NFTMarketplace.sol**

```
27   constructor(IERC20 currency, uint256 listingFee) {
28       _currency = currency;
29       _fee = listingFee;
30   }

     (...SNIP...)

103  function setFee(uint256 fee) public onlyOwner {
104      _fee = fee;
105      emit SetFee(fee);
106  }
```

Listing 8.1 The *constructor* and *setFee* function allowing the developer to set the *platform fee* without limit

## Recommendations

We recommend limiting the scope of *platform fee* in the *constructor* (L27 - 30) and *setFee* function (L103 - 106) so that the developer cannot set the *fee* too high.

For example, we can scope the fee range in between 0 > *fee* <= 100 range as the following code snippet.

**NFTMarketplace.sol**

```
103 function setFee(uint256 fee) public onlyOwner {
104     require(fee > 0, "Fee must be more than 0");
105     require(fee <= 100, "Fee must be less than or equal to 100");
106     _fee = fee;
107     emit SetFee(fee);
108 }
```

Listing 8.2 Example of the *setFee* function with fee scope checks

## Reassessment

The PlayToEarn team fixed this issue by limiting the scope of the *platform fee* as the below code snippet.

**NFTMarketplace.sol**

```
154 function setFee(uint256 _fee) external onlyOwner {
155     uint256 listingFee = _fee.mul(100).div(FEE_DENOMINATOR);
156     require(listingFee >= 0, "Fee must not be less than 0");
157     require(listingFee <= 100, "Fee must not be more than 100");
158     fee = _fee;
159     emit SetFee(listingFee);
160 }
```

Listing 8.3 The fixed *setFee* function

| No. 9 | Possibly Permanent Ownership Removal | | |
|---|---|---|---|
| Risk | Medium | Likelihood | Low |
| | | Impact | High |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *@openzeppelin/contracts/access/Ownable.sol* | | |
| Locations | *Ownable.sol L: 53 - 55* | | |

## Detailed Issue

The *NFTMarketplace* contract inherits from the *Ownable* abstract contract. The *Ownable* contract implements the *renounceOwnership* function, which can remove the ownership of the contract permanently.

If the contract owner mistakenly invokes the *renounceOwnership* function, they will immediately lose ownership of the contract, and this action cannot be undone.

The code snippet below shows the *renounceOwnership* function of the *Ownable* contract.

**Ownable.sol**

```
53  function renounceOwnership() public virtual onlyOwner {
54      _setOwner(address(0));
55  }

    (...SNIP...)

66  function _setOwner(address newOwner) private {
67      address oldOwner = _owner;
68      _owner = newOwner;
69      emit OwnershipTransferred(oldOwner, newOwner);
70  }
```

Listing 9.1 The *renounceOwnership* function that can remove the ownership of the contract permanently

## Recommendations

We consider the *renounceOwnership* function risky, and the contract owner should use this function with extra care.

If possible, we recommend removing or disabling this function from the contract.

## Reassessment

The PlayToEarn team fixed this issue by disabling the *renounceOwnership* function.

**NFTMarketplace.sol**

```
function renounceOwnership() public view override onlyOwner {
    revert("Renounce ownership not allowed");
}
```

Listing 9.2 The disabled *renounceOwnership* function

| No. 10 | Unsafe Ownership Transfer | | |
|---|---|---|---|
| Risk | Medium | Likelihood | Low |
| | | Impact | High |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *@openzeppelin/contracts/access/Ownable.sol* | | |
| Locations | *Ownable.sol L: 61 - 64* | | |

## Detailed Issue

The *NFTMarketplace* contract inherits from the *Ownable* abstract contract. The *Ownable* contract implements the *transferOwnership* function, which can transfer the ownership of the contract from the current owner to another owner.

The code snippet below shows the *transferOwnership* function of the *Ownable* contract.

**Ownable.sol**

```
61  function transferOwnership(address newOwner) public virtual onlyOwner {
62      require(newOwner != address(0), "Ownable: new owner is the zero address");
63      _setOwner(newOwner);
64  }
65
66  function _setOwner(address newOwner) private {
67      address oldOwner = _owner;
68      _owner = newOwner;
69      emit OwnershipTransferred(oldOwner, newOwner);
70  }
```

Listing 10.1 The *transferOwnership* function that has an unsafe ownership transfer

From the code snippet above, the address variable *newOwner* (L61) may be incorrectly specified by the current owner by mistake; for example, an address that a new owner does not own was inputted. Consequently, the new owner loses ownership of the contract immediately, and this action is unrecoverable.

## Recommendations

We recommend applying the two-step ownership transfer mechanism as shown in the code snippet below.

**NFTMarketplace.sol**

```
358 function transferOwnership(address _candidateOwner) external override onlyOwner {
359     require(_candidateOwner != address(0), "Ownable: candidate owner is the zero address");
360     candidateOwner = _candidateOwner;
361     emit NewCandidateOwner(_candidateOwner);
362 }
363
364 function claimOwnership() external {
365     require(candidateOwner == msg.sender, "Ownable: transaction submitter is not the candidate owner");
366
367     address oldOwner = owner;
368     owner = candidateOwner;
369     candidateOwner = address(0);
370     emit OwnershipTransferred(oldOwner, owner);
371 }
```

Listing 10.2 The two-step ownership transfer mechanism

This mechanism works as follows.

1. The current owner invokes the *transferOwnership* function by specifying the candidate owner address *_candidateOwner* (L358).

2. The candidate owner proves access to his account and claims the ownership transfer by invoking the *claimOwnership* function (L364).

The recommended mechanism ensures that the ownership of the contract would be transferred to another owner who can access his account only.

## Reassessment

The PlayToEarn team fixed this issue by applying the two-step ownership transfer mechanism as our recommendation.

| No. 11 | Improper NFT Data Querying | | |
|---|---|---|---|
| Risk | Medium | Likelihood | Medium |
| | | Impact | Medium |
| Functionality is in use | In use | Status | Partially Fixed |
| Associated Files | *(at commit: 6695f55e42a70dc50e7694bbb6ff42de43b7bbf8)*<br>*contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 361 - 396, 416 - 467, and 469 - 506* | | |

## Detailed Issue

*This issue was raised during the reassessment of the issue no. 6 at the commit: 6695f55e42a70dc50e7694bbb6ff42de43b7bbf8.*

The affected functions include:

1. *getMarketItems* function (L361 - 396)
2. *fetchPurchasedNFTs* function (L416 - 467)
3. *fetchCreateNFTs* function (L469 - 506)

We found some querying bugs in the affected functions as follows.

**Bug #1: Overabundant array allocation**

This bug affects only the *getMarketItems* function. We found that the function might allocate memory (L371 - 373 in the code snippet 11.1) for the return variable beyond necessity. The *marketItems* variable allocates memory based on the current number of items selling on the platform (L372) which could be more than the maximum return array elements only limited to 100 (L368).

Subsequently, the function caller (e.g., client) would receive the return data polluted with the empty array elements. In the worst case, moreover, if the number of selling items is too big, the EVM node may refuse to process the query.

**Bug #2: Incorrect paging calculation**

This bug affects all three functions: *getMarketItems*, *fetchPurchasedNFTs*, and *fetchCreateNFTs*. Like the following, the affected functions iterate over items (L374, L428, L443, L481, and L488).

```
for (uint256 i = limit.mul(page).sub(limit); i < limit.mul(page); i++) {
    …
}
```

Consider the following scenario to understand why the paging calculation may be incorrect.

- **1st call:** *getMarketItems*(*page* = 1, *limit* = 5) would return *marketItems*[0, 1, 2, 3, 4] ***(correct item sequence)***
- **2nd call:** *getMarketItems*(*page* = 2, *limit* = 5) would return *marketItems*[5, 6, 7, 8, 9] ***(correct item sequence)***
- **3rd call:** *getMarketItems*(*page* = 3, *limit* = 6) would return *marketItems*[12, 13, 14, 15, 16, 17] ***(incorrect item sequence)***

As you can see, when we change the *limit* from 5 to 6, the function returns the incorrect item sequence.

**Bug #3: Empty return elements**

This bug affects only the *getMarketItems* function. The function iterates over items based on the inputted variables *page* and *limit* as follows.

```
for (uint256 i = limit.mul(page).sub(limit); i < limit.mul(page); i++) {
    if (
        !idToMarketItem[i + 1].isSold &&
        !idToMarketItem[i + 1].isUnlisted &&
        idToMarketItem[i + 1].itemId > 0
        ) {
            ...
        }
}
```

We found that the function would skip the sold-out or unlisted items (L376 - 378), resulting in returning some empty elements to the function caller.

**NFTMarketplace.sol**

```
361  function getMarketItems(uint256 page, uint256 limit)
362      public
363      view
364      returns (MarketItemView[] memory)
365  {
366      require(page > 0, "Page must be more than 0");
367      require(limit > 0, "Limit must be more than 0");
368      require(limit <= 100, "Max limit reached");
369      uint256 currentIndex = 0;
370  
371      MarketItemView[] memory marketItems = new MarketItemView[](
372          itemsSelling.current()
373      );
374      for (uint256 i = limit.mul(page).sub(limit); i < limit.mul(page); i++) {
375          if (
376              !idToMarketItem[i + 1].isSold &&
377              !idToMarketItem[i + 1].isUnlisted &&
378              idToMarketItem[i + 1].itemId > 0
```

```
        ) {
            uint256 currentId = idToMarketItem[i + 1].itemId;
            MarketItemView memory currentItem = MarketItemView({
                itemId: idToMarketItem[currentId].itemId,
                nftContract: idToMarketItem[currentId].nftContract,
                tokenId: idToMarketItem[currentId].tokenId,
                seller: idToMarketItem[currentId].seller,
                price: idToMarketItem[currentId].price,
                amount: idToMarketItem[currentId].amount,
                isSoldOut: idToMarketItem[currentId].isSold,
                isUnlisted: idToMarketItem[currentId].isUnlisted
            });
            marketItems[currentIndex] = currentItem;
            currentIndex += 1;
        }
    }
    return marketItems;
}
```

Listing 11.1 One of the affected functions, *getMarketItems*

## Recommendations

We recommend re-designing/implementing all the affected functions. In addition, we recommend performing unit testing on the functions against all possible edge cases to make sure that the functions return the correct data.

## Reassessment

The PlayToEarn team fixed bugs #1 (*Overabundant array allocation*) and #3 (*Empty return elements*), but bug #2 (*Incorrect paging calculation*) is still effective on the *getMarketItems*, *fetchPurchasedNFTs*, and *fetchCreateNFTs* functions.

The team acknowledged bug #2 and guaranteed not to change the *limit* function parameter when querying data from the front-end.

| No. 12 | No Input Sanitization Checks | | |
|---|---|---|---|
| Risk | Low | Likelihood | Low |
| | | Impact | Medium |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 27 - 30, 103 - 106, and 112 - 115* | | |

## Detailed Issue

It is the recommended practice to validate all input parameters before processing them. As shown in the code snippet below, we found the following functions changing state variables without validating input parameters.

1. *constructor* (L27 - 30)
2. *setFee* function (L103 - 106)
3. *setCurrency* function (L112 - 115)

**NFTMarketplace.sol**

```
 27  constructor(IERC20 currency, uint256 listingFee) {
 28      _currency = currency;
 29      _fee = listingFee;
 30  }

     (...SNIP...)

103  function setFee(uint256 fee) public onlyOwner {
104      _fee = fee;
105      emit SetFee(fee);
106  }

     (...SNIP...)

112  function setCurrency(address currency) public onlyOwner {
113      _currency = IERC20(currency);
114      emit SetCurrency(currency);
115  }
```

Listing 12.1 Functions that change state variables without validating input parameters

## Recommendations

We recommend updating the associated functions to validate all input parameters before processing them.

For example, if the zero address (0) is inputted in the *setCurrency* function, the zero address may lead to unexpected behaviors such as denial of service. Therefore, we recommend validating the zero address in the *setCurrency* function like the below code snippet.

**NFTMarketplace.sol**

```
function setCurrency(address currency) public onlyOwner {
    require(currency != address(0), "Currency must not be the zero address");
    _currency = IERC20(currency);
    emit SetCurrency(currency);
}
```

Listing 12.2 Example of the improved *setCurrency* function with zero address validation check

## Reassessment

The PlayToEarn team fixed this issue by validating all input parameters of the associated functions.

|  |  |  |  |
|---|---|---|---|
| No. 13 | The Compiler Is Not Locked To A Specific Version | | |
| Risk | Low | Likelihood | Low |
| | | Impact | Medium |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 2* | | |

## Detailed Issue

The NFTMarketplace smart contract should be deployed with the compiler version used in the development and testing process.

The compiler version that is not strictly locked via the *pragma* statement may make the contract incompatible against unforeseen circumstances.

The code that is not locked to a specific version (e.g., using `>=` or `^` directive) is shown below.

**NFTMarketplace.sol**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;
```

Listing 13.1 The code that is not locked to a specific version

## Recommendations

We recommend locking the *pragma* version like the example code snippet below.

```
pragma solidity 0.8.0;
// or
pragma solidity =0.8.0;
contract SemVerFloatingPragmaFixed {
}
```

Reference: `https://swcregistry.io/docs/SWC-103`

## Reassessment

The PlayToEarn team fixed this issue by locking the *pragma* version to v0.8.10.

| No. 14 | The Compiler May Be Susceptible To The Publicly Disclosed Bugs | | |
|---|---|---|---|
| Risk | Low | Likelihood | Low |
| | | Impact | Medium |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 2* | | |

## Detailed Issue

The NFTMarketplace smart contract uses an outdated Solidity compiler version which may be susceptible to publicly disclosed vulnerabilities. The compiler version currently used is 0.8.4, which contains the list of known bugs as the following links:

***https://docs.soliditylang.org/en/v0.8.10/bugs.html***

The known bugs may not directly lead to the vulnerability, but it may increase an opportunity to trigger some attacks further.

The smart contract that does not use the latest patch version is shown below.

**NFTMarketplace.sol**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;
```

Listing 14.1 The smart contract that does not use the latest patch version (v0.8.10)

## Recommendations

We recommend using the latest patch version, v0.8.10, that fixes all known bugs.

## Reassessment

The PlayToEarn team fixed this issue by applying the latest Solidity patch version, v0.8.10.

| No. 15 | Recommended Gas Optimization | | |
|---|---|---|---|
| Risk | Informational | Likelihood | Low |
| | | Impact | Low |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 103 - 106, 112 - 115, 117 - 159, 161 - 214, 216 - 250, and 252 - 277* | | |

## Detailed Issue

The following functions can be optimized for saving gas usage by changing their access visibility from *public* to *external*.

1. *setFee* function (L103 - 160)

2. *setCurrency* function (L112 - 115)

3. *createMarketItem* function (L117 - 159)

4. *buyMarketItem* function (L161 - 214)

5. *unlistMarketItem* function (L216 - 250)

6. *setMarketItemPrice* function (L252 - 277)

The code snippet below shows one of the *public* functions that can be optimized for saving gas.

**NFTMarketplace.sol**

```
117  function createMarketItem(
118      address nftContract,
119      uint256 tokenId,
120      uint256 price,
121      uint256 amount
122  ) public nonReentrant {

     (...SNIP...)

159  }
```

Listing 15.1 One of the *public* functions that can be optimized for saving gas

## Recommendations

We recommend changing the access visibility of the associated functions as *external* for gas-saving like the following code snippet.

**NFTMarketplace.sol**

```
function createMarketItem(
    address nftContract,
    uint256 tokenId,
    uint256 price,
    uint256 amount
) external nonReentrant {

(...SNIP...)

}
```

Listing 15.2 The optimized function for saving gas

## Reassessment

The PlayToEarn team fixed this issue by changing the access visibility of the associated functions as *external* for gas-saving.

| **No. 16** | **Misleading Struct Field** | | |
|---|---|---|---|
| **Risk** | **Informational** | **Likelihood** | **Low** |
| | | **Impact** | **Low** |
| **Functionality is in use** | **In use** | **Status** | **Fixed** |
| **Associated Files** | *contracts/NFTMarketplace.sol* | | |
| **Locations** | *NFTMarketplace.sol L: 40* | | |

## Detailed Issue

We found that the *MarketItem* struct has the misleading field *isSold* (L40 in the code snippet below). The *isSold* variable is used to track whether NFT assets under a specific *itemId* are sold out. In other words, the *isSold* variable will be marked as *true* when all NFT assets (under a particular *itemId*) are sold out.

**NFTMarketplace.sol**

```
32  struct MarketItem {
33      uint256 itemId;
34      address nftContract;
35      uint256 tokenId;
36      address seller;
37      address owner;
38      uint256 price;
39      uint256 amount;
40      bool isSold;
41      bool isUnlisted;
42  }
```

Listing 16.1 The *MarketItem* struct with the misleading field *isSold*

## Recommendations

We recommend renaming the associated struct field for clarity, as shown in the code snippet below (L40).

**NFTMarketplace.sol**

```
32 struct MarketItem {
33     uint256 itemId;
34     address nftContract;
35     uint256 tokenId;
36     address seller;
37     address owner;
38     uint256 price;
39     uint256 amount;
40     bool isSoldOut;
41     bool isUnlisted;
42 }
```

Listing 16.2 The improved *`MarketItem`* struct

## Reassessment

The PlayToEarn team fixed this issue according to our recommendation.

| No. 17 | Misleading State Variable | | |
|---|---|---|---|
| Risk | Informational | Likelihood | Low |
| | | Impact | Low |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 19* | | |

## Detailed Issue

The state variable `_itemsSold` is misleading (L19 in the code snippet below). The `_itemsSold` variable is used to track the number of NFT items already sold out (each NFT item can have multiple assets under).

However, the `NFTMarketplace` contract supports buying a partial number of assets (under a specific `itemId`). Therefore, not every purchase transaction will buy all NFT assets.

**NFTMarketplace.sol**

```
13 contract NFTMarketplace is ReentrancyGuard, Ownable, ERC1155Holder {
14     using Counters for Counters.Counter;
15     using SafeERC20 for IERC20;
16     using SafeMath for uint256;
17 
18     Counters.Counter private _itemIds; // Id for each individual item
19     Counters.Counter private _itemsSold; // Number of items sold
20     Counters.Counter private _itemsUnlist; // Number of items delisted
```

Listing 17.1 The `_itemsSold` state variable is misleading

## Recommendations

We recommend renaming the associated state variable for clarity, as shown in the code snippet below (L19).

**NFTMarketplace.sol**

```
13 contract NFTMarketplace is ReentrancyGuard, Ownable, ERC1155Holder {
14     using Counters for Counters.Counter;
15     using SafeERC20 for IERC20;
16     using SafeMath for uint256;
17 
18     Counters.Counter private _itemIds; // Id for each individual item
19     Counters.Counter private _itemsSoldOut; // Number of items sold out
20     Counters.Counter private _itemsUnlist; // Number of items delisted
```

Listing 17.2 The improved state variable

## Reassessment

The PlayToEarn team fixed this issue by renaming the associated state variable as follows.

**NFTMarketplace.sol**

```
contract NFTMarketplace is
  Initializable,
  Ownable,
  ReentrancyGuard,
  ERC1155Holder
{
    using Counters for Counters.Counter;
    using SafeERC20 for IERC20;
    using SafeMath for uint256;

    Counters.Counter private itemIds; // ID for each individual item
    Counters.Counter private itemsSelling; // ID for each individual item
```

Listing 17.3 The renamed state variable

| No. 18 | Inconsistent Comment With The Code | | |
|---|---|---|---|
| Risk | Informational | Likelihood | Low |
| | | Impact | Low |
| Functionality is in use | In use | Status | Fixed |
| Associated Files | *contracts/NFTMarketplace.sol* | | |
| Locations | *NFTMarketplace.sol L: 22* | | |

## Detailed Issue

In L22 of the code snippet below, the state variable *_fee* is used for calculating the commission for the platform owner that is inconsistent with the comment that tells that the commission would be for NFT owners or sellers.

**NFTMarketplace.sol**

```
13 contract NFTMarketplace is ReentrancyGuard, Ownable, ERC1155Holder {
14     using Counters for Counters.Counter;
15     using SafeERC20 for IERC20;
16     using SafeMath for uint256;
17 
18     Counters.Counter private _itemIds; // Id for each individual item
19     Counters.Counter private _itemsSold; // Number of items sold
20     Counters.Counter private _itemsUnlist; // Number of items delisted
21 
22     uint256 private _fee; // This is made for owner of the file to be comissioned (percent)
23 
24     IERC20 private _currency;
25     uint256 private constant FEE_DENOMINATOR = 10**10;
26 
27     constructor(IERC20 currency, uint256 listingFee) {
28         _currency = currency;
29         _fee = listingFee;
30     }
```

Listing 18.1 The inconsistent comment with the source code

## Recommendations

We recommend updating the associated comment to reflect the source code’s transparency.

## Reassessment

The PlayToEarn team fixed this issue by updating the associated comment below.

**NFTMarketplace.sol**

```
contract NFTMarketplace is
  Initializable,
  Ownable,
  ReentrancyGuard,
  ERC1155Holder
{
    using Counters for Counters.Counter;
    using SafeERC20 for IERC20;
    using SafeMath for uint256;

    Counters.Counter private itemIds; // ID for each individual item
    Counters.Counter private itemsSelling; // ID for each individual item

    IERC20 private currency;

    uint256 private fee; // The percentage that game creator will get from each sale
```

Listing 18.2 The improved comment

# Appendix

## About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

## Contact Information

✉ **info@valix.io**

**https://www.facebook.com/ValixConsulting**

**https://twitter.com/ValixConsulting**

**https://medium.com/valixconsulting**

# References

| Title | Link |
|---|---|
| OWASP Risk Rating Methodology | https://owasp.org/www-community/OWASP_Risk_Rating_Methodology |
| Smart Contract Weakness Classification and Test Cases | https://swcregistry.io/ |