NFT Marketplace Audit Report

0xOwain

August 2025

NFT Marketplace Audit Report

Version 1.0

0xOwain

August 2025

Protocol Summary

This project is a basic on-chain NFT marketplace. Users can create NFTs using a simple minting contract (`NFT.sol`), then list them for sale on the marketplace (`Marketplace.sol`). The marketplace lets users: - List NFTs for sale - Purchase listed NFTs - Withdraw proceeds from sales Listings are handled via a local `MarketplaceItem` struct that tracks token ID, price, seller, and buyer. The contract takes a flat listing fee (default: 0.025 ETH) on each transaction.

Disclaimer

0Owain makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Impact Likelihood	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

Repository: https://github.com/igorlourenco/nft-marketplace-smart-contract.git

Commit Hash 16d14cd2a0ecb67f3155c9f47e6a0400de97e57b

Contracts in Scope: - `NFT.sol` - `Marketplace.sol`

cloc Summary

Metric	Value
Contracts Analyzed	2 (`Marketplace.sol`, `NFT.sol`)
Total Lines	231 (Marketplace: 194, NFT: 37)
Total Functions	11

Metric	Value
• Public Functions	9 (`Marketplace`: 6, `NFT`: 3)
• Payable Functions	2 (`createMarketplaceItem()`, `createMarketplaceSale()`)
Cyclomatic Complexity Score	156 (Marketplace: 138, NFT: 18)

Out of Scope:
- N/A

Severity Criteria

High - Direct loss of funds or permanent lock of assets.
- Anyone can exploit (not just privileged roles).
- Breaks core protocol functionality.

Medium
- Causes significant disruption (DoS, griefing, governance failure).
- Exploitable under some conditions or requires privileged roles.
- Financial loss is possible but limited.

Low
- Minor issues: inefficiencies, gas waste, unclear logic, small inconsistencies.
- Doesn't threaten core security or funds.

Informational / Non-Critical
- Code style, readability, missing comments.
- Best practices (naming conventions, event emissions, input validation improvements).
- No security impact.

Summary of Findings

Severity	Number of issues found
High	0
Medium	2
Low	1
Informational	1
Gas Optimisations	0
Total	**4**

Tools Used

- Slither

- Solidity Metrics
- cloc
- visual studio code

Manual Review: A thorough manual review was conducted for both `Marketplace.sol` and `NFT.sol`. Key focus areas included reentrancy vulnerabilities, CEI compliance, gas inefficiencies, and unsafe usage of external calls. Emphasis was placed on logic that handles asset transfers and mutable state updates, especially in `createMarketplaceSale()` and functions dealing with NFT listing and ownership. The audit identified critical control flow patterns that could expose the contract to DoS or future reentrancy bugs, even when current protections (e.g., `nonReentrant`) are in place.

Testing: Custom JavaScript tests with Web3

Static Analysis: The project was statically analyzed using **Slither**, which highlighted several issues:

- **Reentrancy risk** due to state updates occurring after external calls.
- **DoS vulnerability** from the use of `.transfer()` to send ETH.
- **Gas inefficiencies** in `fetchMarketplaceItems()`, `fetchMyNFTs()`, and `fetchItemsCreated()`, due to full iteration over potentially large mappings.
- **Improper immutability** of constant values such as `listingPrice`.

These issues were verified, categorised by severity, and documented with context-aware recommendations for mitigation.

Medium

[M-1] Denial of Service in `Marketplace::createMarketplaceSale()` (Forced Revert on .transfer)

Description: The `Marketplace::createMarketplaceSale()` function pays the seller using Solidity's `.transfer`. This method forwards only 2300 gas and automatically reverts if the recipient's fallback function reverts or requires more gas. If the seller is a contract, they can implement a reverting or gas-hungry fallback. In this case, every attempt to purchase the listed NFT will fail when the marketplace tries to transfer ETH to the seller, rendering the listing permanently unsellable. **Impact:** - Availability risk: Any malicious seller contract can deliberately block the sale of its own NFT, causing denial-of-service for buyers.

- Business impact: Legitimate items may become unsellable, degrading marketplace reliability.
- No direct fund theft: Funds are not stolen, but affected items cannot be sold until the vulnerability is addressed.

Proof of Concept: We created a malicious seller contract whose `receive()` function always reverts when it is sent ETH. The contract was used to mint and list an NFT on the marketplace, making it the seller in the sale. When a buyer attempted to purchase the listed

NFT via `createMarketplaceSale`, the marketplace executed:

```
idToMarketplaceItem[itemId].seller.transfer(msg.value);
```

Because `.transfer` forwards only 2300 gas and reverts on failure, the malicious seller's `receive()` function reverted the transaction. As a result, the entire sale failed, and the NFT remained permanently unsellable.

Malicious Seller (Simplified):

```
contract EvilSeller {
    receive() external payable {
        revert("nope");
    }
}
```

PoC was successfully reproduced in a Hardhat test

Recommended Mitigation: Avoid using `.transfer` to pay the seller. Instead: 1. Preferred approach - Pull Payments: - Record the seller's proceeds in a mapping (e.g., `sellerProceeds[seller] += price`). - Allows sellers to call a `withdrawProceeds()` function to withdraw funds themselves. - This removes the external call from `createMarketplaceSale`, preventing any seller from blocking sales. 2. Alternative approach - Safe `.call`: - Replace `.transfer` with: `solidity (bool success, ) = seller.call{value: msg.value}(""); require(success, "ETH transfer to seller failed");` - This forwards all gas and lets you handle failure gracefully instead of reverting automatically. 3. Additional recommendations: - Always update critical state (`owner`, `sold`) before performing external calls (Checks-Effects-Interactions_). - Ensure the fee transfer to `owner` is handled in the same way.

Severity: Medium Likelihood: Medium Impact: Medium Reference: Slither: Reentrancy

[M-2] CEI Violation in `Marketplace::createMarketplaceSale()` (State Updates After External Calls → Latent Reentrancy Risk)

Description: The `Marketplace::createMarketplaceSale()` function performs two external calls (`seller.transfer` and `IERC721.transferFrom`) before updating critical state variables (`owner` and `sold`). This breaks teh Checks-Effects-Interactions (CEI) pattern, leaving contract state in an inconsistent condition during external interations. In the current codebase, direct recursion into this function is blocked by `nonReentrant`, and other functions touching `idToMarketplaceItem` are `view`, so the issue is not presently exploitable. However, the broken CEI ordering creates a latent reentrancy surface.: if future changes introduce any public/external state-changing entrypoint without `nonReentrant`, a malicious seller contract could reenter mid-sale (during payout) and manipulate marketplace state before `sold` is set.

Impact: - State integrity risk: Mid-execution external calls occur while the item is still logically "unsold", enabling inconsistent reads/writes if a new mutable entrypoint is added later.

- Future exploitability: With an additional unguarded state-changing function, an attacker's fallback could reenter and cause double-actions, incorrect accounting, or ownership/state corruption.
- Current severity rationale: Medium — Likelihood: Low (now), Impact: High (if triggered in a future revision).

Proof of Concept: Although the `createMarketplaceSale()` function is protected by `nonReentrant`, it violates the Checks-Effects-Interactions (CEI) pattern by calling `seller.transfer()` and `nft.transferFrom()` before updating critical state variables such as `owner` and `sold`. This opens the door for reentrancy attacks if: - The `nonReentrant` modifier is ever removed or bypassed. - A new external call is added elsewhere that is not protected. - Other functions using `idToMarketplaceItem` are exposed to external interaction before these variables are updated.

A malicious seller contract could exploit this by triggering arbitrary logic in their fallback during the ETH transfer or during the NFT transfer, possibly reentering a vulnerable function path.

Even if currently unreachable, this ordering creates technical debt and a latent vulnerability surface that could be weaponised if the code evolves.

Recommended Mitigation: 1. Follow the Checks-Effects-Interactions (CEI) pattern: - Move all critical state updates (e.g. setting `owner`, `sold`) before any external calls (`.transfer()`, `transferFrom()`). - This ensures the contract's internal state reflects the sale outcome even if an external call reverts or causes unexpected behavior.

2. Maintain or expand use of `nonReentrant`:
   - Keep the `nonReentrant` modifier on `createMarketplaceSale()` and any future functions that mutate or depend on marketplace state.
3. Defensive Design for Future Development:
   - If new functions are added that call into or depend on idToMarketplaceItem, consider applying reentrancy guards or adhering strictly to CEI in those as well.
   - Avoid making assumptions that current protections (like `nonReentrant`) will cover future complexity.

Severity: Medium Likelihood: Low (currently not exploitable) Impact: High (future exploitability) Reference: Checks-Effects-Interactions

Low

[L-1] Inefficient Iteration in `fetchMarketplaceItems()`, `fetchMyNFTs()`, and `fetchItemsCreated()` Can Lead to High Gas Usage

Description: The functions `fetchMarketplaceItems()`, `fetchMyNFTs()`, and `fetchItemsCreated()` all rely on brute-force iteration over the entire `idToMarketplaceItem` mapping to filter relevant results. Each function performs two full loops: - The first to count matching items. - The second to populate a return array with matching items.

This pattern scales poorly as more items are listed on the marketplace. - `fetchMarketplaceItems()` filters by `owner == address(0)` - `fetchMyNFTs()` filters by `owner == msg.sender` - `fetchItemsCreated()` filters by `seller == msg.sender`

Impact: - Gas usage grows linearly with total number of listings, even if only a few match the criteria. - Risk of exceeding gas limits or timing out for users with large portfolios or marketplaces with long histories. - Poor scalability for future versions or high-volume deployments.

Recommended Mitigation: - Use separate mappings or indexed arrays to track: - Unsold item IDs - User-owned NFTs - User-created listings - Use events for off-chain indexing (e.g. with The Graph) for scalable frontend queries.

Severity: Low Likelihood: High (loop always runs) Impact: Low (limited to gas costs & frontend DoS) Reference: Slither: Unbounded loop

Informational

[I-1] `listingPrice` Should Be Declared `constant` to Reduce Gas and Signal Immutability

Description: In `Marketplace.sol`, the variable `listingPrice` is assigned a fixed value (`0.025 ether`) and is never updated after deployment. Despite its immutability in practice, it is declared as a regular `uint256`, not a `constant`.

Failing to declare such variables as `constant` results in unnecessary storage reads during runtime and obscures the contract's design intent.

```
// Current:
uint256 listingPrice = 0.025 ether;

// Suggested:
uint256 public constant listingPrice = 0.025 ether;
```

Impact: - Gas inefficiency: Each access to `listingPrice` triggers a storage read, which is more expensive than accessing a constant (embedded in bytecode). - Clarity: Using `constant` signals to other developers and tools that the value will never change, improving readability and auditability.

Recommended Mitigation: Change the declaration of `listingPrice` to use the `constant` keyword:

```
uint256 public constant listingPrice = 0.025 ether;
```

This ensures the value is compiled into the bytecode, reducing gas costs and clearly indicating immutability.

Severity: Informational Likelihood: N/A Impact: Low Reference: Slither: Constant variables