CONTRACT WOLF

Security Assessment

LiquidNFT MarketPlace

Verified on 01/06/2026

SUMMARY

Project	CHAIN	METHODOLOGY
LiquidNFT MarketPlace	-	Manual & Automatic Analysis

FILES	DELIVERY	TYPE
Single	01/06/2026	Standard Audit

5	1	1	2	1	0	5
Total Findings	Critical	Major	Medium	Minor	Informational	Resolved

🟥 1 Critical	An exposure that can affect the contract functions in several events that can risk and disrupt the contract
🟧 1 Major	An opening & exposure to manipulate the contract in an unwanted manner
🟧 2 Medium	An opening that could affect the outcome in executing the contract in a specific situation
⬜ 1 Minor	An opening but doesn't have an impact on the functionality of the contract
🟦 0 Informational	An opening that consists information but will not risk or affect the contract
🟩 5 Resolved	ContractWolf's findings has been acknowledged & resolved by the project

STATUS ✔️**AUDIT PASSED**

TABLE OF CONTENTS | LiquidNFT MarketPlace

DISCLAIMER | LiquidNFT MarketPlace

ContractWolf audits and reports should not be considered as a form of project's "Advertisement" and does not cover any interaction and assessment from "Project Contract" to "External Contracts" such as PancakeSwap, UniSwap, SushiSwap or similar.

ContractWolf does not provide any warranty on its released report and should not be used as a decision to invest into audited projects.

ContractWolf provides a transparent report to all its "Clients" and to its "Clients Participants" and will not claim any guarantee of bug-free code within its **SMART CONTRACT**.

ContractWolf's presence is to analyze, audit and assess the Client's Smart Contract to find any underlying risk and to eliminate any logic and flow errors within its code.

Each company or project should be liable to its security flaws and functionalities.

SCOPE OF WORK | LiquidNFT MarketPlace

LiquidNFT MarketPlace team has agreed and provided us with the files that need to be tested (*Github, BSCscan, Etherscan, Local files etc*). The scope of audit is the main contract.

The goal of this engagement is to identify if there is a possibility of security flaws in the implementation of smart contract and its systems.

ContractWolf will be focusing on contract issues and functionalities along with the project claims from smart contract to their website, whitepaper, repository which has been provided by **LiquidNFT MarketPlace**.

AUDITING APPROACH | LiquidNFT MarketPlace

Every line of code along with its functionalities will undergo manual review to check for security issues, quality of logic and contract scope of inheritance. The manual review will be done by our team that will document any issues that they discovered.

METHODOLOGY

The auditing process follows a routine series of steps :

1. Code review that includes the following :
   - Review of the specifications, sources and instructions provided to ContractWolf to make sure we understand the size, scope and functionality of the smart contract.
   - Manual review of code. Our team will have a process of reading the code line-by-line with the intention of identifying potential vulnerabilities, underlying and hidden security flaws.

2. Testing and automated analysis that includes :
   - Testing the smart contract function with common test cases and scenarios to ensure that it returns the expected results.

3. Best practices and ethical review. The team will review the contract with the aim to improve efficiency, effectiveness, clarifications, maintainability, security and control within the smart contract.

4. Recommendations to help the project take steps to eliminate or minimize threats and secure the smart contract.

TOKEN DETAILS | LiquidNFT MarketPlace

-

Token Name	Symbol	Decimal	Total Supply	Chain
-	-	-	-	-

SOURCE

Source	*Sent Via local-files*

FINDINGS | LiquidNFT MarketPlace

5	1	1	2	1	0	5
Total Findings	Critical	Major	Medium	Minor	Informational	Resolved

This report has been prepared to state the issues and vulnerabilities for LiquidNFT MarketPlace through this audit. The goal of this report findings is to identify specifically and fix any underlying issues and errors

ID	Title	File & Line #	Severity	Status
SWC-107	Reentrancy Attack	L: 100	Critical	Resolved
	Bidder Array Management Issues	L: 172	Major	Resolved
	Improper Royalty Handling	L: 152	Medium	Resolved
	Missing Validation	L: 660	Medium	Resolved
SWC-101	Integer Overflow/Underflow	L: 144	Minor	Resolved

SWC ATTACKS | LiquidNFT MarketPlace

Smart Contract Weakness Classification and Test Cases

ID	Description	Status
SWC-100	Function Default Visibility	Passed
SWC-101	Integer Overflow and Underflow	Passed
SWC-102	Outdated Compiler Version	Passed
SWC-103	Floating Pragma	Passed
SWC-104	Unchecked Call Return Value	Passed
SWC-105	Unprotected Ether Withdrawal	Passed
SWC-106	Unprotected SELF DESTRUCT Instruction	Passed
SWC-107	Reentrancy	Passed
SWC-108	State Variable Default Visibility	Passed
SWC-109	Uninitialized Storage Pointer	Passed
SWC-110	Assert Violation	Passed
SWC-111	Use of Deprecated Solidity Functions	Passed
SWC-112	Delegate call to Untrusted Callee	Passed
SWC-113	DoS with Failed Call	Passed
SWC-114	Transaction Order Dependence	Passed
SWC-115	Authorization through tx.origin	Passed
SWC-116	Block values as a proxy for time	Passed
SWC-117	Signature Malleability	Passed
SWC-118	Incorrect Constructor Name	Passed
SWC-119	Shadowing State Variables	Passed
SWC-120	Weak Sources of Randomness from Chain Attributes	Passed
SWC-121	Missing Protection against Signature Replay Attacks	Passed
SWC-122	Lack of Proper Signature Verification	Passed

ID	Description	Status
SWC-123	Requirement Violation	Passed
SWC-124	Write to Arbitrary Storage Location	Passed
SWC-125	Incorrect Inheritance Order	Passed
SWC-126	Insufficient Gas Griefing	Passed
SWC-127	Arbitrary Jump with Function Type Variable	Passed
SWC-128	DoS With Block Gas Limit	Passed
SWC-129	Typographical Error	Passed
SWC-130	Right-To-Left-Override control character(U+202E)	Passed
SWC-131	Presence of unused variables	Passed
SWC-132	Unexpected Ether balance	Passed
SWC-133	Hash Collisions With Multiple Variable Arguments	Passed
SWC-134	Message call with hardcoded gas amount	Passed
SWC-135	Code With No Effects	Passed
SWC-136	Unencrypted Private Data On-Chain	Passed

CW ASSESSMENT | LiquidNFT MarketPlace

ContractWolf Vulnerability and Security Tests

ID	Name	Description	Status
CW-001	Multiple Version	Presence of multiple compiler version across all contracts	✔
CW-002	Incorrect Access Control	Additional checks for critical logic and flow	✔
CW-003	Payable Contract	A function to withdraw ether should exist otherwise the ether will be trapped	✔
CW-004	Custom Modifier	major recheck for custom modifier logic	✔
CW-005	Divide Before Multiply	Performing multiplication before division is generally better to avoid loss of precision	✔
CW-006	Multiple Calls	Functions with multiple internal calls	✔
CW-007	Deprecated Keywords	Use of deprecated functions/operators such as block.blockhash() for blockhash(), msg.gas for gasleft(), throw for revert(), sha3() for keccak256(), callcode() for delegatecall(), suicide() for selfdestruct(), constant for view or var for actual type name should be avoided to prevent unintended errors with newer compiler versions	✔
CW-008	Unused Contract	Presence of an unused, unimported or uncalled contract	✔
CW-009	Assembly Usage	Use of EVM assembly is error-prone and should be avoided or double-checked for correctness	✔
CW-010	Similar Variable Names	Variables with similar names could be confused for each other and therefore should be avoided	✔
CW-011	Commented Code	Removal of commented/unused code lines	✔
CW-012	SafeMath Override	SafeMath is no longer needed starting with Solidity v0.8+. The compiler now has built-in overflow checking.	✔

FIXES & RECOMMENDATION

SWC-107 | Reentrancy

The **_sendValueWithFallback** function and withdraw function are vulnerable to reentrancy attacks:

```
// In withdraw() function - state update after external call
pendingWithdrawals[currency][msg.sender] = 0;  // State update happens too late

// In _sendValueWithFallback()
(success, ) = to.call{value: amount}("");  // External call before state updates
```

Recommendation

Update the state before any external calls

```
function withdraw(address currency) external nonReentrant {
    uint256 amount = pendingWithdrawals[currency][msg.sender];
    if (amount == 0) revert("No funds to withdraw");

    // Update state BEFORE external call
    pendingWithdrawals[currency][msg.sender] = 0;

    if (currency == address(0)) {
        (bool success, ) = msg.sender.call{value: amount}("");
        if (!success) {
            // Revert state change if transfer fails
            pendingWithdrawals[currency][msg.sender] = amount;
            revert TransferFailed();
        }
    } else {
        IERC20(currency).safeTransfer(msg.sender, amount);
    }

    emit Withdrawal(msg.sender, currency, amount);
}
```

SWC-101 | Integer Overflow/Underflow

The **_totalFeeBps** function can overflow due to unchecked addition:

```
function _totalFeeBps(uint96 _royaltyFeeBps, uint96 _liquidFeeBps, uint96
_platformFeeBps)
    internal pure returns (uint256) {
    return uint256(_royaltyFeeBps) + uint256(_liquidFeeBps) +
uint256(_platformFeeBps);
}
```

Recommendation

Use SafeMath or Solidity 0.8.x built-in overflow checks

Improper Royalty Handling

The **`_resolveRoyaltyReceiver`** function returns **`address(0)`** if royalty info is not found, which will cause the **`_distributeProceeds`** function to revert when royalties are due:

```
function _resolveRoyaltyReceiver(address nftContract, uint256 tokenId, uint256
salePrice)
    internal view returns (address receiver) {
    // Returns address(0) if no royalty info
    return address(0);
}
```

Recommendation

```
function _resolveRoyaltyReceiver(address nftContract, uint256 tokenId, uint256
salePrice)
    internal view returns (address receiver) {
    try IERC2981Like(nftContract).royaltyInfo(tokenId, salePrice) returns (address
r, uint256) {
        if (r != address(0)) return r;
    } catch {
        // Return seller as fallback instead of address(0)
    }
    // Fallback: return address(0) but handle in _distributeProceeds
    return address(0);
}

// In _distributeProceeds:
if (royaltyFee > 0) {
    address royaltyReceiver = _resolveRoyaltyReceiver(nftContract, tokenId,
salePrice);
    if (royaltyReceiver == address(0)) {
        // Distribute to seller if no royalty receiver
        remainingAmount += royaltyFee;
    } else {
        _sendValueWithFallback(royaltyReceiver, paymentCurrency, royaltyFee);
    }
}
```

Missing Validation

The **`acceptOffer`** function transfers NFT twice unnecessarily:

```
IERC721(nftContract).safeTransferFrom(msg.sender, address(this), tokenId);
IERC721(nftContract).safeTransferFrom(address(this), offerer, tokenId);
```

Recommendation

Transfer directly from owner to offerer

```
IERC721(nftContract).safeTransferFrom(msg.sender, offerer, tokenId);
```

Bidder Array Management Issues

The bidder array management has inefficient sorting, redundant operations and incorrect refund logic.

- When updating an existing bidder's bid, it doesn't refund their old bid - their old funds stay locked in the contract
- Assumes index 4 is always the lowest bidder - but the array isn't always sorted when checking
- Uses inefficient bubble sort for only 5 elements
- Has redundant double-loops that waste gas

Recommendation

Replace the current implementation with the optimized version (logic sample):

```
function _addOrUpdateBidder(Listing storage listing, address bidder, uint256
bidAmount)
    internal returns (bool success, address refundBidder, uint256 refundAmount) {

    int256 existingIndex = _findBidderIndex(listing, bidder);

    if (existingIndex >= 0) {
        // When updating existing bidder: store their old bid for refund
        uint256 idx = uint256(existingIndex);
        refundBidder = bidder;
        refundAmount = listing.topBidders[idx].bidAmount;
        listing.topBidders[idx].bidAmount = bidAmount;
        success = true;
    } else {
        if (listing.bidderCount < 5) {
            listing.topBidders[listing.bidderCount] = Bidder(bidder, bidAmount);
            listing.bidderCount++;
            success = true;
        } else {
            // Find actual lowest bidder (don't assume index 4)
            uint256 lowestIndex = 0;
            uint256 lowestBid = listing.topBidders[0].bidAmount;

            for (uint256 i = 1; i < 5; i++) {
                if (listing.topBidders[i].bidAmount < lowestBid) {
                    lowestBid = listing.topBidders[i].bidAmount;
                    lowestIndex = i;
                }
            }

            if (bidAmount > lowestBid) {
                refundBidder = listing.topBidders[lowestIndex].bidder;
                refundAmount = lowestBid;
```

```
                listing.topBidders[lowestIndex] = Bidder(bidder, bidAmount);
                success = true;
            } else {
                success = false;
            }
        }
    }

    if (success) {
        // Sort efficiently (insertion sort is better for small arrays)
        _insertionSortBidders(listing);
    }

    return (success, refundBidder, refundAmount);
}

function _insertionSortBidders(Listing storage listing) internal {
    for (uint256 i = 1; i < listing.bidderCount; i++) {
        Bidder memory key = listing.topBidders[i];
        uint256 j = i;
        while (j > 0 && listing.topBidders[j-1].bidAmount < key.bidAmount) {
            listing.topBidders[j] = listing.topBidders[j-1];
            j--;
        }
        listing.topBidders[j] = key;
    }
}
```

CONTRACTWOLF

Blockchain Security - Smart Contract Audits