CONTRACT WOLF

Security Assessment

LiquidNFT Utilities

Verified on 01/06/2026

SUMMARY

Project	CHAIN	METHODOLOGY
LiquidNFT Utilities	Binance Smart Chain	Manual & Automatic Analysis
FILES	**DELIVERY**	**TYPE**
Single	01/06/2026	Standard Audit

	Total Findings	Critical	Major	Medium	Minor	Informational	Resolved
●	3	2	0	0	1	0	3

🟥 2 Critical	An exposure that can affect the contract functions in several events that can risk and disrupt the contract
🟧 0 Major	An opening & exposure to manipulate the contract in an unwanted manner
🟧 0 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
🟩 3 Resolved	ContractWolf's findings has been acknowledged & resolved by the project

STATUS ✔️**AUDIT PASSED**

TABLE OF CONTENTS | LiquidNFT Utilities

Summary

Project Summary
Findings Summary
Disclaimer
Scope of Work
Auditing Approach

Project Information

Token/Project Details
Inheritance Graph
Call Graph

Findings

Issues
SWC Attacks
CW Assessment
Fixes & Recommendation
Audit Comments

DISCLAIMER | LiquidNFT Utilities

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 Utilities

LiquidNFT Utilities 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 Utilities**.

AUDITING APPROACH | LiquidNFT Utilities

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 Utilities

-

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

SOURCE

Source	*Sent Via local-files*

FINDINGS | LiquidNFT Utilities

3	2	0	0	1	0	3
Total Findings	Critical	Major	Medium	Minor	Informational	Resolved

This report has been prepared to state the issues and vulnerabilities for LiquidNFT Utilities 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	LiquidNFTSwapContract	Critical	● Resolved
	Front-Runnable Approve	LiquidVaultMinter	Critical	● Resolved
	Requirement Violation	LiquidVaultMinter	Minor	● Resolved

SWC ATTACKS | LiquidNFT Utilities

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 Utilities

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

Front-Runnable Approve

From : LiquidVaultMinter

The `mint()` function uses `approve(LiquidVault, bal)` without first setting allowance to zero. This allows a front-running attack where the `LiquidVault` could use an old allowance before it's reduced.

Recommendation

Use the safe approve pattern: set to 0 first, then to the new amount.

```
function mint(address to) external override {
    require(isMinter(msg.sender), "LiquidVaultMinter: You are not allowed to mint");
    uint256 bal = IERC20(underlying).balanceOf(address(this));
    require(bal > 0, "LiquidVaultMinter: No Balance");

    // Safe approve pattern: reset to 0 first
    IERC20(underlying).approve(LiquidVault, 0);
    IERC20(underlying).approve(LiquidVault, bal);

    ILiquidVault(LiquidVault).mint(bal, to);

    // Reset approval to 0 after minting for security
    IERC20(underlying).approve(LiquidVault, 0);
}
```

Missing Receiver Validation

From : LiquidVaultMinter

The `mint()` function doesn't validate the address. Tokens could be minted to `address(0)` (burning them) or to contracts that cannot handle ERC20 tokens, permanently locking funds.

Recommendation

Add basic address validation

```
function mint(address to) external override {
    require(to != address(0), "Invalid recipient address");
    require(to != address(this), "Cannot mint to contract itself");
    //
}
```

SWC-107 | Reentrancy

From : LiquidNFTSwapContract

Problem: The `swapAndMint()` function performs external calls to potentially untrusted NFT collections before updating internal state (completing the swap). A malicious NFT collection could re-enter the function and drain funds.

Recommendation

Add reentrancy guard and follow Checks-Effects-Interactions pattern.
Logic Sample :

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract LiquidNFTSwapContract is Ownable, ReentrancyGuard {
    function swapAndMint(
        address collection,
        uint256 mintQty,
        bytes32[] calldata proof,
        address tokenIn,
        uint256 amountIn,
        uint256 minUSDC,
        address to
    ) external payable nonReentrant { // Add nonReentrant modifier
        require(to != address(0), "Invalid recipient");
        require(collection != address(0), "Invalid collection");

        // 1. Swap to FUSD (state changes happen here)
        uint256 fusdReceived = _performSwap(tokenIn, amountIn, minUSDC, msg.value);

        // 2. Check cost (read-only)
        uint256 costPerNFT = INFTMasterCopy(collection)._uints(2);
        uint256 totalCost = costPerNFT * mintQty;
        require(fusdReceived >= totalCost, "Insufficient FUSD");

        // 3. Transfer FUSD to user first, let them approve and mint separately
        // OR keep current flow but ensure no reentrancy
        IERC20(fusdToken).safeTransfer(msg.sender, fusdReceived);

        emit SwapCompleted(msg.sender, fusdReceived, totalCost);

        // Note: User now calls NFT collection directly with their FUSD
        // This separates swap logic from minting logic
    }
```

```
 // Alternative: Keep minting but add strict checks
    function swapAndMintV2(...) external payable nonReentrant {
        // swap logic

        // Verify collection contract
        require(_isContract(collection), "Not a contract");
        require(INFTMasterCopy(collection).mintToken() == fusdToken, "Token mismatch");

        // Transfer FUSD to THIS contract temporarily
        uint256 remainingFUSD = fusdReceived;

        // APPROACH: Use a pull payment pattern instead of approve
        _transferAndMint(collection, mintQty, proof, totalCost, to);

        // Refund remaining
        if (remainingFUSD > 0) {
            IERC20(fusdToken).safeTransfer(msg.sender, remainingFUSD);
        }
    }
}
```

No Issues | Clean Contract

From : Generator, Database

ContractWolf did not find any technical issues within the contract and marked the contract safe to interact with.

CONTRACTWOLF

Blockchain Security - Smart Contract Audits