QuillAudits

Audit Report, March, 2024

For

NFTFN

Table of Content

Table of Content

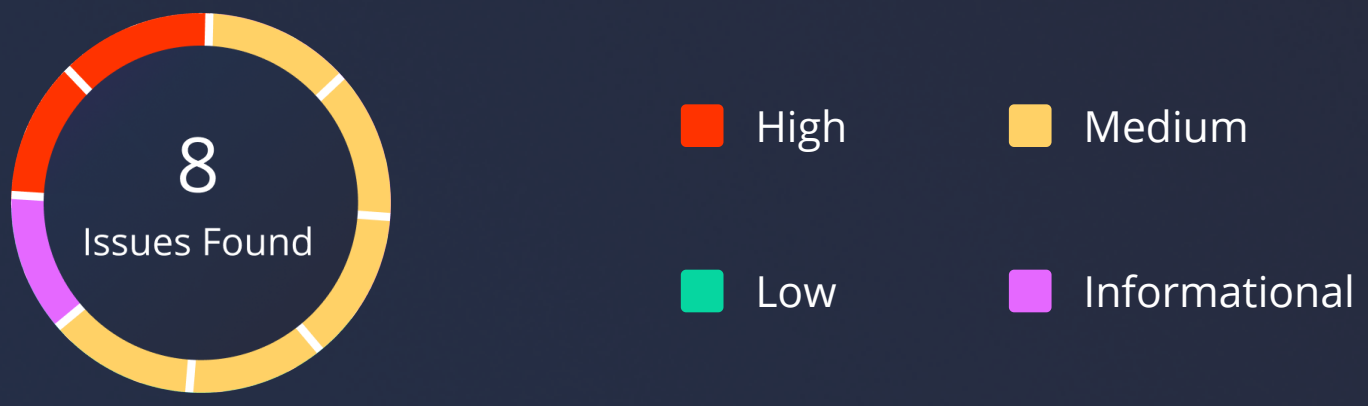
Executive Summary

Project Name	NFTFN
Project URL	https://www.nftfn.xyz
Overview	NFTFN is a web3 fintech company focused on creating financial products leveraging NFTs. The goal of NFTFN is to create a suite of products around NFTs and solving the liquidity challenge around this new emerging market.
Audit Scope	*https://sepolia.etherscan.io/address/0x825181Cc677D66B581311a5a0e3b3101C9456189#code* *https://sepolia.etherscan.io/address/0x81a7cd9e8ff668f016f682410afe7885daa89b30#code*
Contracts in Scope	Contracts:- 1] Presale contract: A simple Presale Contract is meant to sell tokens in stages where the amount of token allocated and the USD rate are determined before starting the stage and remain constant 2]Token Contract: Standard ERC20 token with an admin based mechanism for minting and burning tokens
Commit Hash	NA
Language	Solidity
Blockchain	Ethereum
Method	Manual Review, Automated tools,Functional testing
Review 1	27th Feb 2024 - 5th Mar 2024

Executive Summary

Updated Code Received	6th Mar 2024
Review 2	6th March 2024 - 7th March 2024
Fixed In	Zip File Shared by NFTFN Team https://drive.google.com/drive/folders/1jagoj0w8MifZaSBLA5QNWulELR_zNziZ?usp=share_link

Number of Issues per Severity



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	2	4	0	1

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw

Checked Vulnerabilities

- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle,Solhint, Mythril, Slither, Solidity static analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

A. Contract - NFTFN.sol

High Severity Issues

A.1 Restrictive Token Burning Functionality

Line	Function - burn

57-63	function burn(address from, uint256 amount) public onlyAdmin {
	if (from != msg.sender) {
	revert InvalidAddress();
	}
	
	_burn(from, amount);
	}

Description

The current implementation of the burn function within the smart contract restricts the ability to burn tokens unnecessarily by enforcing that only the admin can initiate a token burn, and exclusively for their own tokens. This restriction is imposed by the combination of the onlyAdmin modifier and a conditional check ensuring the from address matches the msg.sender. This design not only limits the utility and flexibility of the token burn mechanism but also potentially hampers token economics management and user engagement strategies.

Remediation

Reassess the requirement that only admins can burn tokens, and consider allowing users to burn their own tokens. This empowers token holders with greater control over their assets and can enable dynamic token economics.

Status

Resolved

B. Contract - PreSale.sol

High Severity Issues

B.2 Lack of ETH Refund Calculation in **refundTokenAllocation** Function

Line **Function - refundTokenAllocation**

389-419

```
function refundTokenAllocation(address user, address token) public {
        /// Check that Soft Limit has not been reached
        if (totalUsdRaised >= usdSoftLimit) revert SoftLimitHit();

        /// Revert If Token is Invalid
        if (!acceptedTokens[token]) revert InvalidToken();

        /// Revert if the stage has not been active for 30 days or the stage has no
start time i.e. it was never started
        if (presaleParamsData[currentStage].startTime == 0 ||
presaleParamsData[currentStage].startTime + (30 * 86400) > block.timestamp
        ) revert RefundNotActive(); // if not able to achieve target in 30 days,
then refund.

        UserData storage userData = userDeposits[user];

        uint256 usdInvested = userData.usdInvested;

        /// Check that the user has an allocation that he is asking a refund for
        if (usdInvested == uint256(0)) revert InvalidBalance();

        /// If the amount need to refund is greated than the amount currently
held in the stage

        /// Remove his invested and tokens allocated
        totalUsdRaised -= usdInvested;
        presaleParamsData[currentStage].usdRaised -= usdInvested; // @audit
why we are dedudting the overall amount of an user from current stage.
        presaleParamsData[currentStage].tokensAllocated +=
userData.tokensAllocated;
```

```
                userData.tokensAllocated = 0;
                userData.usdInvested = 0;

                IERC20Metadata(token).safeTransfer(user, usdInvested);

                emit RefundClaimed(user, block.timestamp);
        }
```

Description

The refundTokenAllocation function in the presale contract allows users to refund their token allocations if certain conditions are met, such as the soft limit not being reached within a specified timeframe. While the function supports refunds for token purchases made with accepted ERC-20 tokens (e.g., USDT), it lacks the mechanism to calculate and process refunds for purchases made with ETH. This oversight can lead to a scenario where users who participated in the presale using ETH cannot receive their refunds in ETH, thereby affecting the fairness and integrity of the presale contract.

Remediation

Implement ETH Refund Logic: Modify the refundTokenAllocation function to include logic that calculates and processes refunds in ETH for users who contributed in ETH. This involves tracking the payment method (ETH or ERC-20 token) at the time of investment and the corresponding USD value of the ETH contributed, considering the ETH/USD exchange rate at the time of purchase.

Status

Resolved (Intended functionality)

Medium Severity Issues

B.3 Flawed Accounting in **refundTokenAllocation** Function

Line	Functions - refundTokenAllocation

389 - 419

```
function refundTokenAllocation(address user, address token) public {
        /// Check that Soft Limit has not been reached
        if (totalUsdRaised >= usdSoftLimit) revert SoftLimitHit();

        /// Revert If Token is Invalid
        if (!acceptedTokens[token]) revert InvalidToken();

        /// Revert if the stage has not been active for 30 days or the stage has no
start time i.e. it was never started
        if (presaleParamsData[currentStage].startTime == 0 ||
presaleParamsData[currentStage].startTime + (30 * 86400) > block.timestamp
        ) revert RefundNotActive(); // if not able to achieve target in 30 days,
then refund.

        UserData storage userData = userDeposits[user];

        uint256 usdInvested = userData.usdInvested;

        /// Check that the user has an allocation that he is asking a refund for
        if (usdInvested == uint256(0)) revert InvalidBalance();

        /// If the amount need to refund is greated than the amount currently
held in the stage

        /// Remove his invested and tokens allocated
        totalUsdRaised -= usdInvested;
        presaleParamsData[currentStage].usdRaised -= usdInvested; // @audit
why we are dedudting the overall amount of an user from current stage.
        presaleParamsData[currentStage].tokensAllocated +=
userData.tokensAllocated;
        userData.tokensAllocated = 0;
        userData.usdInvested = 0;

        IERC20Metadata(token).safeTransfer(user, usdInvested);

        emit RefundClaimed(user, block.timestamp);
 }
```

Description

1. The refundTokenAllocation function in the presale contract is designed to handle refunds for users under certain conditions. However, there is an accounting flaw in how refunds are processed, particularly in how the total USD invested by a user is subtracted from the USD raised in the current presale stage. The function deducts the user's total investment from presaleParamsData[currentStage].usdRaised, without considering that the user's total investment might span multiple stages of the presale. This approach can lead to inaccurate accounting of funds raised in the current stage, potentially affecting the operation and fairness of the presale process

Remediation

1. **Stage-wise Investment Tracking:** Enhance the data structure to track user investments stage-wise rather than aggregating them into a single total. This will allow for precise adjustments to the funds raised per stage during refunds.

2. **Adjust Refund Logic:** Update the refundTokenAllocation function to correctly account for the stage-specific investment of the user. Deduct the refund amount from the stage in which the funds were actually raised.

Status

Resolved

B.4 Unverified Oracle Price Freshness in **getLatestPrice**

Line	Function - getLatestPrice

493-498	function getLatestPrice() public view returns (uint256) {
	/// This gives answer in 8 decimals
	(, int256 answer,,,) = ethDataFeed.latestRoundData();
	if (answer < 0) revert InvalidPriceFeedValue();
	return uint256(answer * (10 ** 10)); // Conversion to 18 decimals
	}

Description

The function getLatestPrice fetches the latest price from an oracle but does not check whether the price is stale. While the function does convert negative responses into a revert condition indicating an invalid price feed value, it lacks validation to ensure that the price data is recent. In the context of blockchain applications, especially those involving financial transactions, relying on outdated price information can lead to incorrect valuations, exploitation, and potentially significant financial losses.

Remediation

1. **Timestamp Verification:** Modify the getLatestPrice function to include a check for the timestamp of the latest round data. The oracle typically provides this timestamp, which should be compared against the current block timestamp to ensure the data is within an acceptable freshness threshold.

2. **Define Acceptable Delay:** Establish a constant or configurable parameter within the contract that defines the maximum acceptable delay (in seconds) for price data. This parameter allows the contract to reject oracle data that is older than this threshold, mitigating the risk of using stale information.

Status

Resolved

B.5 Unauthorized Refunds using the refundTokenAllocation

Line **Functions - refundTokenAllocation**

389 - 419

```
function refundTokenAllocation(address user, address token) public {
        /// Check that Soft Limit has not been reached
        if (totalUsdRaised >= usdSoftLimit) revert SoftLimitHit();

        /// Revert If Token is Invalid
        if (!acceptedTokens[token]) revert InvalidToken();

        /// Revert if the stage has not been active for 30 days or the stage has no
start time i.e. it was never started
        if (presaleParamsData[currentStage].startTime == 0 ||
presaleParamsData[currentStage].startTime + (30 * 86400) > block.timestamp
        ) revert RefundNotActive(); // if not able to achieve target in 30 days,
then refund.

        UserData storage userData = userDeposits[user];

        uint256 usdInvested = userData.usdInvested;

        /// Check that the user has an allocation that he is asking a refund for
        if (usdInvested == uint256(0)) revert InvalidBalance();

        /// If the amount need to refund is greated than the amount currently
held in the stage

        /// Remove his invested and tokens allocated
        totalUsdRaised -= usdInvested;
        presaleParamsData[currentStage].usdRaised -= usdInvested; // @audit
why we are dedudting the overall amount of an user from current stage.
        presaleParamsData[currentStage].tokensAllocated +=
userData.tokensAllocated;
        userData.tokensAllocated = 0;
        userData.usdInvested = 0;

        IERC20Metadata(token).safeTransfer(user, usdInvested);

        emit RefundClaimed(user, block.timestamp);
 }
```

Description

The refundTokenAllocation function within the contract permits any caller to initiate a refund for any user without requiring explicit authorization from the user whose funds are being refunded. This functionality poses a significant security risk, as it enables potential attackers or unauthorized parties to drain funds from the presale stages by triggering refunds on behalf of legitimate users without their consent. This flaw undermines the integrity of the fundraising process and the security of participant funds.

Remediation

Authentication Requirement: Modify the refundTokenAllocation function to require that the caller is either the user requesting the refund or an authorized admin. This can be achieved through a modifier that checks the caller's address against the intended refund recipient or a list of authorized accounts.

Status

Resolved

B.6 Unauthorized Withdrawals by Owner Before Presale Ends

Line	Function - forwardFunds

424-427	function forwardFunds(address token, uint256 value) public onlyOwner {
	IERC20Metadata(token).safeTransfer(treasuryWallet, value);
	emit FundsTransferred(token, value);
	}

Description

The forwardFunds function in the smart contract allows the contract owner to transfer tokens to a specified treasury wallet without any restriction on the presale's status. This means the owner can withdraw funds (including tokens collected during the presale) at any time, even before the presale concludes. Such functionality undermines the trust of participants in the presale's integrity and security, potentially enabling premature or unauthorized access to the funds raised, contrary to the expected lock-in period or conditions communicated to presale participants.

Remediation

- **Implement NFTFN token check:** Amend the forwardFunds function to include a condition that checks whether the token address is NFTFN token and if it voilates the condition the function should revert.

Status

Resolved

Low Severity Issues

No issues were found.

B.7 Owner can renounce using renounceOwnership function

Line	Function - PreSale Contract
9	import { Ownable } from "../../lib/openzeppelin-contracts/contracts/access/Ownable.sol";

Description

The Owner of the contract is usually the account that deploys the contract. As a result, the Owner is able to perform some privileged functions like initPresale(), initNextPresaleStage(), endPreSale(), forwardFunds() and forwardFundsEth() etc. In the PreSale.sol smart contract, the renounceOwnership function is used to renounce the Owner permission. Renouncing ownership before transferring would result in the contract having no Owner, eliminating the ability to call privileged functions.

Remediation

It is recommended that the Owner is not able to call renounceOwnership without transferring the Ownership to another address before. In addition, if a multi-signature wallet is used, calling renounceOwnership function should be confirmed for two or more users. As another solution, Renounce Ownership functionality can be disabled.

Status

Acknowledged

Low Severity Issues

No issues were found.

Informational Issues

B.8 Typographical Error in Function Name

Line	Function - getClaimiableTokens
370	getClaimiableTokens()

Description

The function getClaimiableTokens contains a typographical error in its name (Claimiable instead of Claimable).

Remediation

Rename the function to getClaimableTokens to correct the typographical error and improve code readability.

Status

Resolved

Functional Tests

Some of the tests performed are mentioned below:

- Should initiate the contract with provided USDC address
- Should initiate the contract with provided WhitelistManager address
- Should initiate the contract with provided owner address
- Should Set the contract Variables by Owner
- Should not Set the contract Variables by other than Pool Manager
- Should not deposit if not whitelisted
- Should add Whitelist
- Should deposit if whitelisted
- Should deposit with locking duration
- Should check Base APR
- Should check Active Deposits
- Should withdraw the funds related to deposit ID EMERGENCY
- Should withdraw by ID but Restriction of Time
- Should withdraw all funds
- Should transfer funds to another account
- Should Set the contract Variables by Owner
- Should not Set the contract Variables by other than Pool Manager
- Should not deposit if not whitelisted
- Should revert withdraw if deposit ID is invalid
- Should revert withdraw if locking duration is not over
- Should revert deposit if token address is invalid or user not whitelisted
- Should revert deposit if locking duration if user not whitelisted or amount 0
- Should set contract variables by owner

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the NFTFN Project. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in **NFTFN PreSale** smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of **NFTFN PreSale** smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services.. It is the responsibility of the **NFTFN PreSale** to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
March, 2024

For

NFTFN

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com