SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

For

NFTFN

Audited By
Aliasgar Kapadia

Report Prepared By
Aliasgar Kapadia

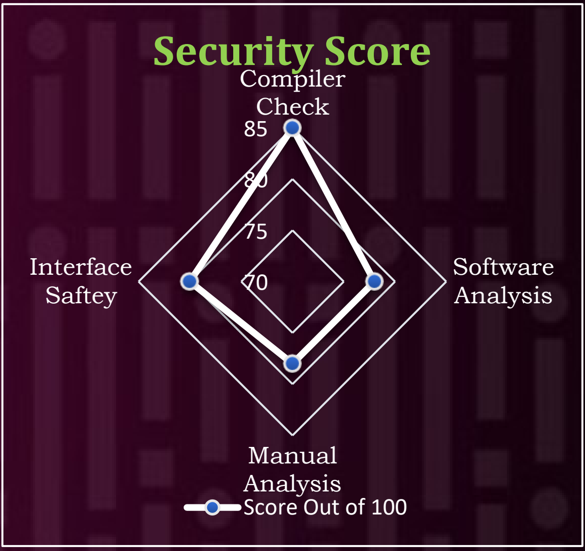
Table of Contents

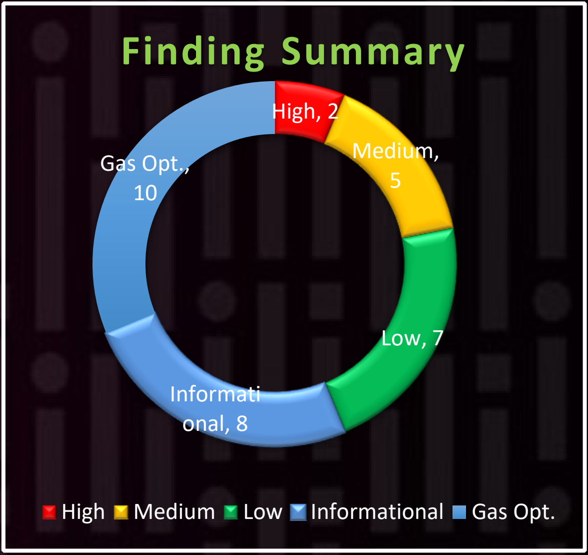
Disclaimer

The Secureverse team examined this smart contract in accordance with industry best practices. We made every effort to secure the code and provide this report. audits done by smart contract auditors and automated algorithms; however, it is crucial to remember that you should not rely entirely on this report. The smart contract may have flaws that allow for hacking. As a result, the audit cannot ensure the explicit security of the audited smart contracts. The Secureverse and its audit report do not encourage readers to consider them as providing any project-related financial or legal advice.

Executive Summary

Project Name	NFTFN
Project Type	DeFi, Dex
Audit Scope	Check security and code quality
Audit Method	Manual
Audit Timeline	21st Sept., 2023 to 29th Sept. 2023
Source Code	NFTFN-FloorPerp-SmartContract.zip





Issue Tracking Table

	High	Medium	Low	Informational	Gas Optimization
Open Issues	-	-	-	-	-
Acknowledged Issues	1	3	2	-	3
Resolved Issues	1	2	5	8	7

Types of Severities

- **High:** The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.

- **Medium:** The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.

- **Low:** The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.

- **Informational:** The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth.

- **Gas Optimization:** The issue also does not pose an immediate risk, but it is the process to making smart contracts more efficient, cost-effective, to enhance scalability and better user experience.

Types of Issues

- **Open:** Security vulnerabilities identified that must be resolved and are currently unresolved.

- **Acknowledged:** The way in which it is being used in the project makes it unnecessary to address the vulnerabilities. This means that the way it has been acknowledged has no effect on its security.

- **Resolved:** These are the issues identified in the initial audit and have been successfully fixed.

Checked Vulnerabilities

- ❖ Re-entrancy
- ❖ Access control
- ❖ Denial of service
- ❖ Timestamp Dependence
- ❖ Integer overflow/Underflow
- ❖ Transaction Order Dependency
- ❖ Requirement Violation
- ❖ Functions Visibility Check
- ❖ Mathematical calculations
- ❖ Dangerous strict equalities
- ❖ Unchecked Return values
- ❖ Hard coded information
- ❖ Safe Ether Transfer
- ❖ Gas Consumption
- ❖ Incorrect Inheritance Order
- ❖ Centralization
- ❖ Unsafe external calls
- ❖ Business logic and specification
- ❖ Input validation
- ❖ Incorrect Modifier
- ❖ Missing events
- ❖ Assembly usage
- ❖ ERC777 hooks
- ❖ Token handling

Methods

Audit at Secureverse is performed by the experts and they make sure that audited project must comply with the industry security standards.

Secureverse audit methodology includes following key:

- In depth review of the white paper
- In depth analysis of project and code documentation.
- Checking the industry standards used in Code/Project.
- Checking and understanding Core Functionality of the Code.
- Comparing the code with documentation.
- Manual analysis of the code.
- Gas Optimization and Function Testing.
- Verification of the overall audit.
- Report writing.

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

Manual Analysis

Manual analysis is done by our smart contract auditors' team by performing in depth analysis of the smart contract and identify potential vulnerabilities. Auditor also review and verify all the static analysis results to prevent the false positives identified by automated tools.

Gas Consumption and Function Testing

Function testing done by auditors by manually writing customized test cases for the smart contract to verify the intended behavior as per code and documentation. Gas Optimization done by reviews potential gas consumption by contract in production.

Findings

High Severity Issues:

[H-01] Potential loss of funds in `recieveDeposit()`

Reference: ProtocolVault/InsuranceFund#L83-L95

Description:

The *recieveDeposit()* function in the *InsuranceFund* contract is designed to receive deposits in tokens other than *WETH*, swap them for *WETH*, and return any dust amount back to the sender. However, there are critical issue in this function that can result in a potential loss of funds. There can be 2 issues arise due to this function:

1) Missing Token Transfer: The function currently does not transfer tokens from the sender's account to the contract. This means that if the contract contains tokens other than *WETH*, and if by chance the *UniswapRouter* contract contains these tokens, the contract may attempt to swap tokens that do not belong to the user.

2) Transferring Dust Amount: After swapping or attempting to swap, the function checks the balance of the contract and transfers any balance to the sender. This approach could lead to a significant issue. If the contract holds tokens other than those deposited by the user, these tokens may be transferred to the sender instead.

This issue can lead to unintended token transfers and potential financial loss for users who interact with this function. Users may receive tokens they did not deposit, and tokens held by the contract may be transferred to users.

Recommendation:

```
function recieveDeposit(address _tokenAddress, uint256 _amount) public whenNotPaused
{
    require(_amount != 0, "Amount can not be zero.");
    if (_tokenAddress != WETH) {
        uint256 balanceBefore = IERC20(_tokenAddress).balanceOf(address(this));
        IERC20(_tokenAddress).safeTransferFrom(msg.sender, address(this), _amount);
        uint256 __amount = _trySwapTokens(_tokenAddress, _amount);
        uint256 balanceAfter = IERC20(_tokenAddress).balanceOf(address(this));
        //Sending out the the dust back

        if ((balanceAfter - balanceBefore) > 0) {
            IERC20(_tokenAddress).safeTransferFrom(address(this), msg.sender,
balanceAfter);
        }
    }
    emit Events.InsuranceFundDepositReceived(_tokenAddress, __amount);
}
```

Status: Resolved

[H-02] Admin can remove itself leading protocol without admin

Reference: Utils/Admin.sol#L26-L31

Description:
The *removeAdmin()* in the Admin contract allows an admin to remove itself from the list of admins. This could potentially lead to a situation where the contract has no admins, making it impossible to manage the admin list and resulting in unexpected behavior.

If the last admin removes itself, the contract will be left without any admin control. This situation can lead to a complete halt of functionalities that only admins should perform, potentially causing a severe disruption in contract operations. Furthermore, any funds or assets that can only be withdrawn by admins will become permanently inaccessible, leading to a significant financial risk for users.

Recommendation:
Consider preventing admins from removing themselves from the admin list. You can do this by adding a condition that checks if the address being removed is the sender's address and, if so, revert the transaction.

```
function removeAdmin(address _addr) public onlyAdmin {
    if (_addr == address(0)) revert Errors.ZeroAddress();

    // Prevent admins from removing themselves
    require(_addr != msg.sender, "Admin cannot remove itself.");

    admins[_addr] = false;
    emit Events.AdminRemoved(_addr);
}
```

Status: Acknowledged

Medium Severity Issues:

[M-01] Hardcoded open order limit needs to be dynamic

Reference: Market/OrderBook.sol#L173

Description:

In *addMakerOrder()* of *OrderBook* contract, there's a line of code that checks whether the number of open orders for a user is greater than or equal to *5*:
if (openOrders[msg.sender].length >= 5)

However, there is a variable named *allowedOpenOrderCount* in the contract, which represents the allowed count of open orders. This value can be changed by an admin using the *updateAllowedOpenOrderCount()*.

The problem here is that hardcoded value "5" is being used instead of using the dynamic value from *allowedOpenOrderCount*. This means that even if the *allowedOpenOrderCount* changes, the contract will continue to enforce the outdated and hardcoded open order limit of *5*.

If the admin changes the *allowedOpenOrderCount* to a different value, the contract will not enforce this new limit. This could lead to a situation where users can create more open orders than intended, potentially causing system instability, resource exhaustion, and unfair trading conditions.

Recommendation:

To make the contract more flexible and adaptable to changes in the allowed open order count, replace the hardcoded value *5* with the dynamic variable *allowedOpenOrderCount*. This ensures that the open order limit is always in sync with the value set by the admin and avoids discrepancies.

```
if (openOrders[msg.sender].length >= allowedOpenOrderCount) {
            revert Errors.OpenOrderCountExceeded();
        }
```

Status: Resolved

[M-02] Centralization risk

Description:

The current setup of the project grants extensive authority to the admin role, allowing them to control critical functions that influence the core functionality of the system. If the admin account were to be compromised, it could lead to severe vulnerabilities and potential exploitation. Below functions are handled by `onlyAdmin`:

- Margin.sol
  - `setOrderBook()`
  - `updateMakerFees()`
  - `updateTakerFees()`
  - `updateIMF()`
  - `updateMMF()`
  - `updateLiquidationPenalty()`
  - `updateProtocolFeesShare()`
  - `updateDepositLimit()`
  - `bringOverWater()`
- OrderBook.sol
  - `setMarginLedger()`
  - `updateQuoteLot()`
  - `updateAllowedOpenOrderCount()`
  - `addAuthorizedAddress()`
  - `removeAuthorizedAddress()`
  - `setOracle()`
  - `initMarkPrice()`
- InsuranceFund.sol
  - `setProtocolVault()`
  - `updateUniswapRouter()`
  - `addFees()`
- Oracle.sol
  - `addNodes()`
- ProtocolVault.sol
  - `setMarginContract()`
  - `setOrderBookContract()`
  - `addFees()`
  - `updateUniswapRouter()`
  - `depositToInsuranceFund()`
  - `withdrawAdmin()`
- Admin.sol
  - `setAdmin()`
  - `removeAdmin()`
- `_pause()` and `_unpause()` in all related contracts.

Recommendation:

Explore the implementation of a TimeLock contract as the protocol owner, enabling users to oversee and understand proposed changes before they are executed. Alternatively, consider transferring the admin role to a governance-controlled address, promoting community involvement and transparency in decision-making processes.

Status: Acknowledged

[M-03] Lack of Access control

Reference: **ProtocolVault/InsuranceFund.sol#L83**

Description:
In the *recieveDeposit()* function, there's a lack of access control, allowing external access without restrictions. This means anyone can call the function. If tokens are mistakenly transferred to this account. Without proper access controls, tracking the source of the funds and recording individual deposits becomes challenging. According to the documentation, only *ProtocolVault* should have permission to invoke this function for depositing funds.

Recommendation:
Use *onlyProtocolVault* modifier to restrict external usage.

Status: **Acknowledged**

[M-04] Return value of *transfer()/transferFrom()* not checked

Reference:
1) **ProtocolVault/InsuranceFund.sol#L91**
2) **ProtocolVault/ProtocolVault.sol#L97, L165, L178**

Description:
There are some instances found that do not check the return value of *transfer/transferFrom* (some tokens signal failure by returning false instead of revert). It has been observed that there are specific instances within the contract where token transfers exclusively involve *WETH* token. However, the concerns raised in issues related to transfers involving tokens provided by users. This issue do not directly apply to transfers involving *WETH* tokens.

Recommendation:
It is good to add a *require()* statement that checks the return value of token transfers or to use OpenZeppelin's *safeTransfer*/*safeTransferFrom* unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

Status: **Resolved**

[M-05] Lack of slippage tolerance in can lead to front-running and fund loss

Reference: UniswapRouter/UniswapRouter.sol#L-56, L58

Description:
The current implementation of the *_trySwapTokens()* has two issues that expose users to significant risks:

1) Slippage Tolerance: This function lacks a slippage tolerance mechanism, which poses a significant risk to user's funds. The issue arises when the *amountOutMinimum* is hardcoded to *0* in *Router.exactInputSingle()*, allowing for any level of slippage during token swaps. In volatile markets or when dealing with large orders, the price can shift while the transaction is being mined, causing the actual received token amount to be less than expected.

2) No User-Controlled Deadline: Contract does not allow users to specify their own transaction deadline. Instead, it uses *block.timestamp* as the deadline, which effectively means there is no set deadline for the transaction. Transactions with no specified deadline may remain pending in the mempool, creating an opportunity for attackers or MEV bots to manipulate the transaction order and potentially conduct sandwich attacks.

Recommendation:
Implement a slippage tolerance mechanism and accept a user-defined deadline parameter or use a specific deadline after the *block.timestamp*.

Status: **Acknowledged**

Low Severity Issues:

[L-1] *setAdmin()* should be 2-step process

Reference: Utils/Admin.sol#L19-L25

Description:
Lack of two-step procedure for critical operations (like add a new admin) leaves them error-prone. The basic validation whether the address is not a zero address is performed, however the case when the address receiving the role is inaccessible is not covered properly.

Recommendation:
Implement a two-step admin addition process:

1. The existing admin nominates a new admin candidate using the *setAdmin()*
2. The new admin candidate accepts the nomination using an *acceptAdminNomination()*.
3. After accepting the nomination, the candidate becomes a full admin.

```
struct AdminCandidate {
    bool exists;
    bool accepted;
}
mapping(address => AdminCandidate) private adminCandidates;
mapping(address => bool) public admins;

function setAdmin(address _newAdmin) public onlyAdmin {
    if (_newAdmin == address(0)) revert Errors.ZeroAddress();
    adminCandidates[_newAdmin].exists = true;
    adminCandidates[_newAdmin].accepted = false;
    emit Events.AdminNominated(_newAdmin);
}

function acceptAdminNomination() public {
    require(adminCandidates[msg.sender].exists, "No admin nomination found for this
address");
    require(!adminCandidates[msg.sender].accepted, "You have already accepted admin
nomination");

    adminCandidates[msg.sender].accepted = true;
    admins[msg.sender] = true;
    emit Events.NewAdminAdded(msg.sender);
}
```

Status: Acknowledged

[L-2] Missing zero-address and values check in constructors and the setter functions

Reference:

1) Margin/Margin.sol#L53, L116
2) Market/OrderBook#L98, 102, 119
3) Oracle/Oracle.sol#L31-L32, L42
4) ProtocolVault/InsuranceFund.sol#L39, L47
5) ProtocolVault/ProtocolVault.sol#L29, L32-L33, L60, L66, L80

Description:

Missing checks for zero-addresses and zero value may lead to unfunctional protocol, if the variable addresses and values are updated incorrectly. While the `Admin` contract has effectively addressed this issue but other than that this practice is not followed. Whilst for some of these variables zero value may be valid, for others it may result in an unexpected, potentially malicious behavior.

It's worth observing that many setter functions in the contract utilize the `onlyAdmin` modifier, ensuring they can only be called by authorized individuals. However, there exists a potential vulnerability where an admin might inadvertently add `address(0)`. To enhance security, it's advisable to include a check for the zero address and values before assigning addresses and values.

Recommendation:

Consider adding zero-address and values checks in the constructors and setter functions. For zero-address the recommended approach outlined in issue [G-07] can be utilized.

Status: **Acknowledged**

[L-03] Missing event for critical functions

Reference:

1) Market/OrderBook#L97, 101
2) ProtocolVault/ProtocolVault.sol#L120, L161

Description:

Functions that change critical contract parameters/addresses/state should emit events so that users and other privileged roles can detect upcoming changes (by off-chain monitoring of events).

Status: **Resolved**

[L-04] Unused Internal functions

Reference:
1) Utils/Base.sol#L72, L76, L80, L84
2) Utils/EnumerableArray.sol#L41, L60, L78, L103, L136, L155
3) ProtocolVault/InsuranceFund.sol#L103

Description:
The contract contains an internal function that is not called or referenced anywhere within the contract's code. This unused function doesn't contribute to the contract's functionality and can be safely removed to maintain a cleaner and more efficient codebase. This unused function does not serve any purpose and only adds unnecessary complexity to the code.

Status: Resolved

[L-05] Use of deprecated `safeApprove()`

Reference: ProtocolVault/InsuranceFund.sol#L103-L111

Description:
The contract currently uses a deprecated *safeApprove()* function from *OpenZeppelin's ERC20* library. It is crucial to be aware that using this deprecated function can result in unintended issues, including potential reverts and the risk of funds becoming inaccessible. A comprehensive discussion regarding the deprecation of this function can be found in OpenZeppelin's issue ***#2219***.

Status: Resolved

[L-06] Should `approve(0)` first

Reference: UniswapRouter/UniswapRouter.sol#L41

Description:
_trySwapTokens() relies on the *approve()* function to manage token approvals, which is a common practice for interacting with ERC20 tokens. Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. The issue arises when these non-standard tokens are utilized as assets within a Uniswap pool. The contract does not handle these tokens correctly due to their non-standard behavior.

Recommendation:
Approve with a zero amount first before setting the actual amount.

Status: Resolved

[L-07] Missing check for contract existence

Reference: **UniswapRouter/UniswapRouter.sol#L41**

Description:
The low-level calls are being used without verifying the existence of the target contract. A low-level call executed on a non-existent contract will return success, even though the intended contract does not exist on the blockchain. More information can be find here: https://docs.soliditylang.org/en/v0.8.7/control-structures.html#error-handling-assert-require-revert-and-exceptions

Recommendation:
Check before any low-level call that the address actually exists, for example before the low level call in the function you can check that the address is a contract by checking its code size:
`<address>.code.length > 0`

Status: **Resolved**

Informational Issues

[NC-01] The variable names for constants do not adhere to the Solidity style guide:

Reference:

1) Margin/Funding.sol#L17
2) Forwarder/Forwarder.sol#L21, L27, L38, L41, L42, L43
3) Utils/Base.sol#L10, L11, L12, L13, L14

Recommendation:

Name of the constant should be UPPER CASE.

Status: Resolved

[NC-02] *`public`* functions not called internally should be declared *`external`*

Reference: ProtocolVault/ProtocolVault.sol#L120

Status: Resolved

[NC-03] Spelling error.

Reference: Margin/Margin.sol#L535

Status: Resolved

[N-04] *`require()`*/*`revert()`* statements should have descriptive reason strings

Reference: UniswapRouter/UniswapRouter.sol#L43

Status: Resolved

[N-05] Unresolved TODOs

Reference:

1) UniswapRouter/UniswapRouter.sol#L45
2) Forwarder/Forwarder.sol#L40, L58

Status: Resolved

[N-06] Unused *events* and *errors*

Reference:

1) Utils/Errors.sol#L18, L26, L37, L44, L48, L58, L80
2) Utils/Events.sol#L20, L22, L28, L53, L60, L61

Status: Resolved

[N-07] Lack of ReentrancyGuard

Reference: ProtocolVault/ProtocolVault.sol#120

Description:
In the *withdraw()* of the *ProtocolVault* contract, there is a potential security vulnerability related to reentrancy attacks. The function relies on an external call to *_tryTransferAsset()* with *msg.sender* as a parameter. This external call creates an opportunity for reentrancy attacks, where an attacker can manipulate the flow of execution to potentially exploit funds.

While the current implementation follows Checks-Effects-Interactions (CEI) pattern, but any minor changes to the code could inadvertently introduce significant security vulnerabilities. Therefore, there is a potential risk associated with this implementation.

Recommendation:
Consider adding the *nonReentrant* modifier to the *withdraw()*.

Status: Resolved

[N-08] Unused imports

Reference:

1) Margin/BalanceMath.sol#L9
2) Margin/Funding.sol#L9
3) Margin/Margin.sol#L5, L23
4) Market/Market.sol#L7
5) UniswapRouter/UniswapRouter.sol#L6

Status: Resolved

Gas Optimization

[G-01] `x = x + y` is cheaper than `x += y`

Reference:

1) Margin/BalanceMath.sol#L131, L132, L210, L211, L227, L230,
2) Market/OrderBook.sol#L339, L340

Status: Acknowledged

[G-02] Should not perform a lookup for `<array>.length` within each iteration of a for-loop

Reference:

1) Margin/BalanceMath.sol#L207
2) Margin/Margin.sol#L287
3) Market/OrderBook.sol#L308, 337, 472, 523

Recommendation:

Optimizing the loop by storing the array's length in a variable before entering it can significantly reduce gas consumption. In scenarios where the length is fetched from memory, this approach can save approximately 3 gas per iteration. Thus, it's recommended to cache the array's length in a variable and use this variable within the loop for better efficiency.

Status: Resolved

[G-03] Use the `constant` keyword for unchanging variables

Reference: Margin/Funding.sol#L14, L18

Description:

Variable can be declared as `constant` if its value remains unchanged throughout its usage. This not only helps in code clarity but also optimizes gas consumption by reducing storage operations.

Status: Resolved

[G-04] Internal functions only called once can be in-lined to save gas

Reference:

1) Margin/BalanceMath.sol#L168, L180
2) Margin/Margin.sol#L103, L107
3) Market/Market.sol#L54
4) Market/OrderBook.sol#L221, L309, L331, L374, L473
5) ProtocolVault/ProtocolVault.sol#L141, L152
6) Utils/Base.sol#L18, L91
7) Utils/EnumerableArray.sol#L8, L29

Description:

Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

Status: **Acknowledged**

[G-05] Redundant check for `address(0)` in function with `onlyAdmin` modifier

Reference: Margin/Margin.sol#114

Description:

No need check against `address(0)` for `msg.sender` within a function that has the `onlyAdmin` modifier. The `onlyAdmin` modifier already guarantees that the function can only be called by addresses that have been explicitly designated as administrators. Therefore, there is no need to include this check for `address(0)` in the function. The `onlyAdmin` modifier ensures that only authorized administrators can access the function, and this check doesn't provide any additional security or functionality. Removing this unnecessary check can lead to more efficient and gas-effective code execution.

Status: **Resolved**

[G-06] Avoid explicitly initializing variables to default values

Reference:

1) Margin/BalanceMath.sol#206
2) Market/OrderBook.sol#L19, L308, L375, L472, L523
3) Margin/Margin.sol#L287
4) Utils/Oracle.sol#L15

Description:
Explicitly initializing a variable with its default value, such as *0* for *uint*, *false* for *bool*, or *address(0)* for address when it's not set/initialized, is considered an anti-pattern and results in unnecessary gas consumption.

Status: Resolved

[G-07] Use assembly to check for *address(0)*

Description:
It's generally more gas-efficient to use assembly to check for a zero address *(address(0))* than to use the == operator.

The reason for this is that the == operator generates additional instructions in the EVM bytecode, which can increase the gas cost of the contract. By using assembly, you can perform the zero-address check more efficiently and reduce the overall gas cost of the contract.

Here's an example of how to use assembly to check for a zero address:

```
function isZeroAddress(address addr) public pure returns (bool) {
    uint256 addrInt = uint256(uint160(addr));

    assembly {
        // Load the zero address into memory
        let zero := mload(0x00)

        // Compare the address to the zero address
        let isZero := eq(addrInt, zero)

        // Return the result
        mstore(0x00, isZero)
        return(0, 0x20)
    }
}
```

Status: Resolved

[G-08] Check for empty array

Reference: Market/OrderBook.sol#L257-L303
Description:
Gas can be saved by checking empty *counterOrderIds[]* in *addTakerOrder()* of *OrderBook* contract. Currently, the function includes a check against the maximum number of counter order IDs allowed. While this is essential for security, there's also an additional optimization that can be applied.

Specifically, by checking for an empty *counterOrderIds* array at the beginning of the function, gas consumption can be reduced. This is because performing this check early on can prevent unnecessary execution of subsequent logic when *counterOrderIds* is empty.

Recommendation:

```
function addTakerOrder(bytes memory _tradeData) public whenNotPaused onlyAuthorized
{
    (address taker, uint256 amount, bool isBuy, uint256[] memory counterOrderIds,
uint256 nextOrderId) =
        abi.decode(_tradeData, (address, uint256, bool, uint256[], uint256));
    // Sanity and Verification
    if (taker == address(0)) revert Errors.ZeroAddress();
    if (amount % getQuoteLot() != 0 || amount == 0) {
        revert Errors.InvalidOrder();
    }
    if (counterOrderIds.length == 0) {
        revert Errors.InvalidCounterOrders();
    }
    //code
}
```

Status: Resolved

[G- 09] Reduce storage gas costs by avoiding bool variables

Reference:

1) Market/OrderBook.sol#L34
2) Oracle/Oracle#L24
3) Utils/Admin#L8

Description:

Storing Boolean values in Solidity contracts can be relatively expensive due to the additional gas costs associated with read-modify-write operations. Each write operation for a Boolean emits an extra SLOAD, which can significantly impact gas consumption, especially in contracts with frequent state changes.

Recommendation:

Use uint256(0) for false and uint256(1) for true. This can eliminate the extra SLOAD operations, significantly optimizing gas usage.

Status: Acknowledged

[G-10] Gas can be saved by declaring constant as private rather than public

Reference:

4) Margin/BalanceMath.sol#L15
5) Margin/Fundin.sol#L17
6) Utils/Structs.sol#L5
7) Utils/Base.sol#L10-L14

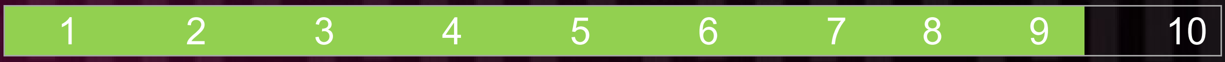
Description:

Constants do not need to be publicly accessible, as their values are fixed and don't change during contract execution. Declaring them as public variables incurs unnecessary gas costs when accessing them, especially if they are read frequently.

Status: **Resolved**

Closing Summary

In this audit, we examined the NFTFN’s smart contract with our framework, and we discovered several High, Medium, Low, Informational and Gas Optimizations flaws in the smart contract. We have included solutions and recommendations in the audit report to improve the quality and security posture of the code. All of the findings and solutions have been acknowledged by the project team. In summary, we find that the codebase with the latest version greatly improved on the initial version. We believe that the overall level of security provided by the codebase in its current state is reasonable, so we have marked it as **Secure** and the customer's smart contract has the following score: 9



About Secureverse

Secureverse is the Singapore and India based emerging Web3 Security solution provider. We at Secureverse provides the Smart Contract audit, Blockchain infrastructure Penetration testing and the Cryptocurrency forensic services with very affordable prices.

To Know More

Twitter: https://twitter.com/secureverse

LinkedIn: https://www.linkedin.com/company/secureverse/

Telegram: https://t.me/secureverse

Email Address: secureverse@protonmail.com