Infinity Funds

Smart Contract Security Audit

Prepared by Nexarias

March 15th, 2024 - April 8th, 2024

Nexarias.com

contact@nexarias.com

Document Properties

Client	Ledger of Infinity
Version	0.2
Classification	Private

Scope

The Infinity Funds Contract in the Infinity Funds Repository

Link	MD5 Hash
https://mumbai.polygonscan.com/address/0x320fc64e6153234797a1231fb1d495963dddbd52	c048a34c74741e7c0bec1a3bdaf12774

Contacts

COMPANY	CONTACT
Nexarias	contact@nexarias.com

Contents

1 Introduction

Ledger of Infinity engaged Nexarias to conduct a security assessment on the Infinity Funds beginning on March 13th, 2024 and ending April 8th, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Infinity Funds

Issuer	Ledger of Infinity
Website	www.ledgerofinfinity.com
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

Nexarias used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by Nexarias are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

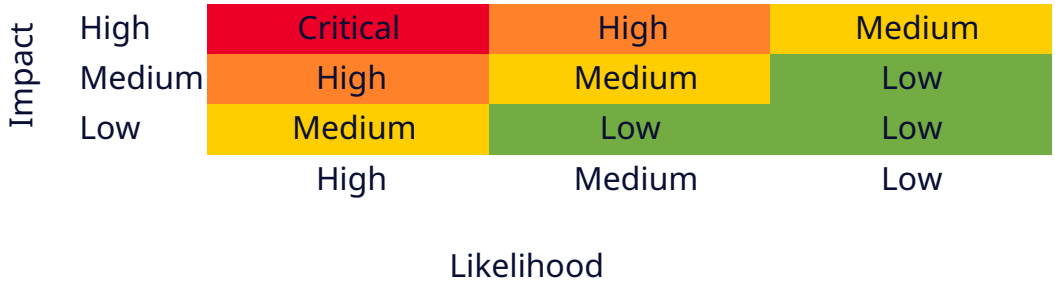
Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.

- Impact quantifies the technical and economic costs of a successful attack.

- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.



2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Infinity Funds implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 2 medium-severity, 1 informational-severity vulnerabilities.

Vulnerabilities	Severity	Status
Use of Outdated ERC20 Implementation	MEDIUM	Not Fixed
ERC20: Insufficient Balance Check in _approve	MEDIUM	Fixed
Integer Overflow and Underflow	INFORMATIONAL	Not Fixed

3 Finding Details

A IFF.sol

A.1 Use of Outdated ERC20 Implementation [MEDIUM]

Description:

The contract implements its own version of the ERC20 standard, which may not include the latest security practices, optimizations, and features found in widely-used libraries such as OpenZeppelin's ERC20 implementation. Using an outdated or custom implementation can introduce risks and compatibility issues with other contracts and decentralized applications (dApps).

Recommendation:

Replace the custom ERC20 implementation with the latest version of the ERC20 token standard from a reputable library such as OpenZeppelin. This not only ensures compliance with the latest security practices but also benefits from the community's scrutiny, ongoing maintenance, and updates.

Status - Not Fixed

A.2 ERC20: Insufficient Balance Check in _approve [MEDIUM]

Description:

The _approve function includes a check require(_balances[owner] > amount, "ERC20: insufficient balance"), which is not a standard requirement for the approval process in ERC20 tokens. This restriction may limit the usability of the token by preventing users from approving more tokens than they currently hold, which is a common practice in some DeFi protocols.

Code:

Listing 1: IFF.sol

```
125 function _approve(address owner, address spender, uint256 amount)
        ,→ private {
126 require(owner != address(0), "ERC20: approve from the zero address");
127 require(spender != address(0), "ERC20: approve to the zero address");
128 require(_balances[owner] > amount, "ERC20: insufficient balance");
129 _allowances[owner][spender] = amount;
130 emit Approval(owner, spender, amount);
131 }
```

Recommendation:

Consider removing the balance check in the _approve function to adhere to standard ERC20 behavior, allowing users to approve an amount regardless of their current balance.

Status - Fixed

A.3 Integer Overflow and Underflow [INFORMATIONAL]

Description:

The contract utilizes a library, SafeMath, for uint256 operations to prevent overflow and underflow vulnerabilities. However, starting from Solidity 0.8.0, arithmetic operations revert on overflow and underflow by default, making the use of SafeMath redundant in this context.

Code:

Listing 2: IFF.sol

```
16 library SafeMath {
```

Recommendation:

Consider removing the SafeMath library to simplify the contract and reduce gas costs, as Solidity version 0.8.0 and above inherently prevents these issues.

Status - Not Fixed

4 Best Practices

BP.1 Explicit Visibility in Constructor

Description:

The constructor in the IFF contract is implicitly public. Since Solidity 0.5.0, specifying the visibility of constructors is not required as they are assumed to be public. However, explicitly declaring visibility improves code readability and understanding.

Code:

Listing 3: IFF.sol

```
74 constructor() public {
75     _mint(msg.sender, 840000000 * 10**uint(decimals));
76 }
```

BP.2 Code Organization and Readability

Description:

The contract's code lacks clear organization and separation of concerns, making it challenging to read and audit. Proper organization, including segmenting functionality into logical sections and using comments to describe the purpose of functions and variables, can significantly enhance readability and maintainability.

5 Static Analysis (Slither)

Description:

Block Hat expanded the coverage of the specific contract areas using automated testing methodologies.Slither, a Solidity static analysis framework,was one of the tools used. Slither was run on all-scoped contracts in both text and binary formats.This tool can be used to test mathematical relationships between Solidity instances statically and variables that allow for the detection of errors or inconsistent usage of the contracts' APIs throughout the entire codebase.

Results:

```
Compilation warnings/errors on .\IFF.sol:
Warning: SPDX license identifier not provided in source file. Before
    ,→ publishing, consider adding a comment containing "SPDX-License-
    ,→ Identifier: <SPDX-License>" to each source file. Use "SPDX-
    ,→ License-Identifier: UNLICENSED" for non-open-source code. Please
    ,→ see https://spdx.org for more information.
--> IFF.sol

Warning: Visibility for constructor is ignored. If you want the contract
    ,→ to be non-deployable, making it "abstract" is sufficient.
  --> IFF.sol:74:1:
   |
74 | constructor() public {
   | ^ (Relevant source part starts here and spans across multiple lines
       ,→ ).


INFO:Detectors:
Context._msgData() (IFF.sol#11-14) is never used and should be removed
SafeMath.div(uint256,uint256) (IFF.sol#38-40) is never used and should
    ,→ be removed
```

```
SafeMath.div(uint256,uint256,string) (IFF.sol#41-45) is never used and
    ,→ should be removed
SafeMath.mod(uint256,uint256) (IFF.sol#46-48) is never used and should
    ,→ be removed
SafeMath.mod(uint256,uint256,string) (IFF.sol#49-52) is never used and
    ,→ should be removed
SafeMath.mul(uint256,uint256) (IFF.sol#30-37) is never used and should
    ,→ be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ,→ #dead-code
INFO:Detectors:
Pragma version^0.8.24 (IFF.sol#5) necessitates a version too recent to
    ,→ be trusted. Consider deploying with 0.8.18.
solc-0.8.24 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ,→ #incorrect-versions-of-solidity
INFO:Detectors:
Variable IFF._totalSupply (IFF.sol#71) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ,→ #conformance-to-solidity-naming-conventions
INFO:Detectors:
Redundant expression "this (IFF.sol#12)" inContext (IFF.sol#6-15)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ,→ #redundant-statements
INFO:Detectors:
IFF.constructor() (IFF.sol#74-76) uses literals with too many digits:
        - _mint(msg.sender,840000000 * 10 ** uint256(decimals)) (IFF.sol
            ,→ #75)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ,→ #too-many-digits
INFO:Detectors:
IFF.decimals (IFF.sol#70) should be constant
IFF.name (IFF.sol#69) should be constant
IFF.symbol (IFF.sol#68) should be constant
```

```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ,→ #state-variables-that-could-be-declared-constant
INFO:Slither:.\IFF.sol analyzed (4 contracts with 93 detectors), 14
    ,→ result(s) found
```

Conclusion:

Most of the vulnerabilities found by the analysis have already been addressed by the smart contract code review.

6 Conclusion

We examined the design and implementation of Infinity Funds in this audit and found several issues of various severities. We advise Ledger of Infinity team to implement the recommendations contained in all 3 of our findings to further enhance the code's security. It is of utmost priority to start by addressing the most severe exploit discovered by the auditors then followed by the remaining exploits, and finally we will be conducting a re-audit following the implementation of the remediation plan contained in this report.

We would much appreciate any constructive feedback or suggestions regarding our methodology, audit findings, or potential scope gaps in this report.

For a Smart Contract Audit, contact us at contact@nexarias.com