Security Review Report
NM-0214-ORA-IMO

(Jun 12, 2024)

Contents

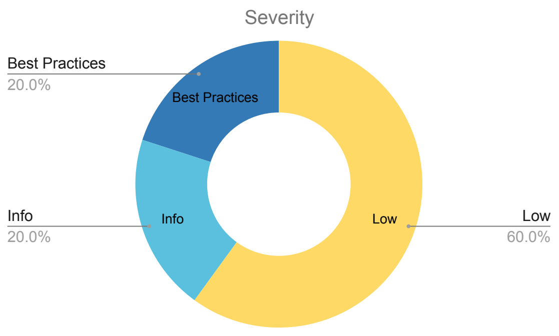
1 Executive Summary

This document outlines the security review conducted by Nethermind for Initial Model Offering developed by ORA. ORA aims to bring AI and complex computing on-chain. The audited product `Initial Model Offering (IMO)` launches and distributes the new token standard `ERC7641`. Holders of this token can claim the revenue generated by the AI model. This solution gives an opportunity for open-source AI models to compete with products created by closed-source, for-profit companies since contributors can be rewarded by holding the token and claiming revenue.

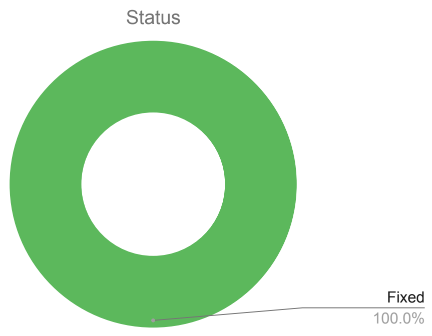
The audited code comprises 139 lines of code. The `ORA` team has provided documentation explaining the behavior of the token design and distribution. Aside from the provided documentation, the `ORA` and `Nethermind` teams have actively communicated to clarify any remaining questions about the expected behavior of the protocol.

The audit was performed using: (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts. **Along this document, we report** 5 points of attention, where three are classified as `Low`, and two are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (3), **Undetermined** (0), **Informational** (1), **Best Practices** (1). **(b) Distribution of status: Fixed** (5), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Apr 16, 2024
Final Report	Jun 12, 2024
Methods	Manual Review, Automated Analysis
Repository	IMOContract-Audit
Commit Hash	3a4ebaf6f8046185f7c49b55ea31514a93292d25
Final Commit Hash	9f38615b0bd2f65bc87ae2882845ddb9b99ad5b5
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/IIMO.sol	12	10	83.3%	9	31
2	src/IMO.sol	127	4	3.1%	29	160
	Total	**139**	**14**	**10.1%**	**38**	**191**

3 Summary of Issues

	Finding	Severity	Update
1	Lack of input validation	Low	Fixed
2	Unreacheable logic within `buyableIMOAmount(...)` function	Low	Fixed
3	Users might lose funds when purchasing IMO tokens with zero contract balance	Low	Fixed
4	Incorrect update of `userBuySpent` value	Info	Fixed
5	Unused variables	Best Practices	Fixed

4 System Overview

The **Initial Model Offering (IMO)** contract launches a revenue sharing token and allows users to buy it for either `ETH` or other defined `ERC20` token. Each address is allowed to spend up to `10**18` `ETH` or `ERC20` token. Users are only allowed to buy tokens with the correct signature to ensure that all tokens are bought through the official website. The contract uses the following storage variables:

```
address public imoToken;
address public buyToken;
uint256 public buyStartTime;
uint256 public buyEndTime;
uint256 public tokenPrice;
address public signer;
uint256 public constant MIN_PER_BUY = 0.1 ether;
uint256 public constant MAX_PER_USER = 3 ether;
uint256 public constant DECIMAL = 10 ** 18;
address public constant ETH_IDENTIFIER = address(0);
mapping(address => uint256) public userBuySpent;
```

The contract exposes the following public or external functions:

Constructor function

```
constructor(...);
```

Buys the token for ETH or ERC20 token

```
function buyIMOToken(...) external payable
```

Returns the amount of IMO tokens buyable for a given amount

```
function buyableIMOAmount(...) public view
```

Returns sum of the amount spent by given addresses

```
function getBuySpentByUsers(...) external view
```

Returns the amount of IMO tokens left to buy

```
function getIMORemaining(...) public view
```

Sets the time period to buy IMO token

```
function setBuyPeriod(...) external
```

Sets the IMO token address

```
function setIMOTokenAddress(...) external
```

Sets the address of the token used to buy

```
function setBuyTokenAddress(...) external
```

Sets initial IMO token price

```
function setTokenPrice(...) external
```

Sets the signer address

```
function setSigner(...) external
```

Transfers collected funds to the owner

```
function collectBuyToken(...) external
```

Transfers IMO tokens

```
function moveIMOTokens(...) external
```

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Low] Lack of input validation

File(s): `IMO.sol`

Description: Several important variables within the `IMO` contract are initialized with values that lack proper validation. We list below the variables and concerned functions:

- `buyEndTime`: There's a lack of validation to ensure that the value assigned to `buyEndTime` exceeds the current `block timestamp`. This validation should be enforced in both the contract constructor and the `setBuyPeriod(...)` function;
- `imoToken`, `tokenPrice`, `signer`: These variables must not be assigned `0` addresses or values in the contract constructor or the associated setter functions: `setIMOTokenAddress(...)`, `setTokenPrice(...)`, and `setSigner(...)`;

Recommendation(s): Consider implementing checks to validate the values assigned to important variables.

Status: Fixed

Update from the client: addressed in `72c46058b4a034e7ac493a596292cd086d4ec954`

Update from Nethermind Security: The `setBuyPeriod(...)` function should not allow the case where `_buyEndTime == block.timestamp`, this case means that the `_buyStartTime` is in the past.

Update from the client: addressed in 9f38615b0bd2f65bc87ae2882845ddb9b99ad5b5

6.2 [Low] Unreacheable logic within `buyableIMOAmount(...)` function

File(s): `IMO.sol`

Description: The `buyableIMOAmount(...)` function serves to calculate the amount of `IMO` tokens a user can buy. For that, it checks whether the provided `buyAmount`, when combined with the previously spent amount, exceeds the allowed spending limit for the user `MAX_PER_USER`. If the total surpasses the allowed limit, the function updates the purchase amount to the remaining allowed amount and refunds the excess to the user.

However, this intended logic is currently unreachable due to a preceding `require` statement. This requirement ensures that the sum of `buyAmount` and the amount already spent by the user doesn't exceed the `MAX_PER_USER` threshold. If this threshold is exceeded, the function reverts.

```
function buyableIMOAmount(...) public view returns (...) {
    // @audit The function reverts if the accumulated amount exceeds `MAX_PER_USER`
    require(buyAmount + userBuySpent[msg.sender] <= MAX_PER_USER, "User accumulated buy token amount exceed maximum");
    uint256 buyAllowed = MAX_PER_USER - userBuySpent[msg.sender];
    uint256 buyAmountToSpend = buyAmount;

    // @audit This `if` statement cannot be reached
    if (buyAmount > buyAllowed) {
        buyAmountToSpend = buyAllowed;
        refund = buyAmount - buyAllowed;
    } else {
        refund = 0;
    }
// ...
}
```

Recommendation(s): Consider removing the `require` statement to allow the execution of the logic within the `if` statement.

Status: Fixed

Update from the client: addressed in 6cdd3e33e88ee0db3fa306b1ff92421ad0080abc

Update from Nethermind Security: The described issue is solved by refactoring the function. However, those changes applied to the buy mechanism introduced another issue. The `_transferInAnyToken()` function doesn't ensure that `msg.value` is equal to 0 when the token is not native ETH. If a user mistakenly sends Ether when calling `buyIMOToken(...)`, he loses access to these funds. Additionally, they will be locked in the contract as the `collectBuyToken()` function won't allow collecting Ether. Consider adding a check that `msg.value == 0` when the token is different from `ETH_IDENTIFIER`.

6.3 [Low] Users might lose funds when purchasing tokens with zero contract balance

File(s): `IMO.sol`

Description: When users attempt to purchase IMO tokens, they specify the amount of the `buyToken` they want to spend. This amount is then converted into an equivalent amount of IMO tokens using the fixed price. However, because this conversion involves division by the price, it can lead to rounding errors. While these errors are usually unavoidable and neglected when purchasing IMO tokens, they lead to strange behavior when the contract's balance of IMO tokens is zero.

Users can still execute the `buyIMOToken(...)` function even when the contract holds no IMO tokens. In this scenario, the function logic will refund the user the total amount provided. However, due to potential rounding errors in the conversion process, the user might receive a slightly smaller refund than expected while no IMO tokens are purchased.

```
function buyableIMOAmount(uint256 buyAmount) public view returns (...) {
  // ...
  uint256 buyAmountToSpend = buyAmount;
  // ...
  //@audit Possible rounding error
  imoAmount = buyAmountToSpend * DECIMAL / tokenPrice;
  uint256 remainingIMOAmount = imoToken.balanceOf(address(this));

  if (imoAmount > remainingIMOAmount) {
      // @audit a total refund is given if `remainingIMOAmount` is zero
      uint256 excessAmount = imoAmount - remainingIMOAmount;
     // @audit the computed `refund` can be slightly lower than the initial `imoAmount`
      refund += excessAmount * tokenPrice / DECIMAL;
      imoAmount = remainingIMOAmount;
  }
}
```

Recommendation(s): Consider implementing additional checks to handle the scenario where the contract's balance of IMO tokens is zero. This can be done by either reverting or returning the exact amount of funds the user provided as a refund.

Status: Fixed

Update from Nethermind Security: The applied changes to the `buyableIMOAmount(...)` protect user from buying zero IMO tokens due to rounding error. However, the described case where contracts does not have any IMO tokens is not fixed, and user can be refunded with possibly less amount than they paid. Consider reverting when the contract does not have any IMO tokens.

Update from the client: addressed in 9f38615b0bd2f65bc87ae2882845ddb9b99ad5b5

6.4 [Info] Incorrect update of `userBuySpent` value

File(s): `IMO.sol`

Description: The `userBuySpent` mapping keeps track of the total amount of `buyToken` spent by each user. It is increased during each buy operation by the spent amount. Within the `_buyIMO(...)` function, when a user attempts to buy a certain `buyAmount`, they may only spend a portion of it, with the function refunding the remaining amount. However, this refund logic is not considered when updating the `userBuySpent` mapping. The function increases the mapping by the total `buyAmount` amount, even though only `buyAmount - refund` is actually spent. This can result in recording a spent amount that is higher than the predefined limit `MAX_PER_USER`.

```
function _buyIMO(uint256 buyAmount) private {
  (uint256 imoAmount, uint256 refund) = buyableIMOAmount(buyAmount);

  //@audit mapping is updated with `buyAmount`, however only `buyAmount - refund` is actually spent
  userBuySpent[msg.sender] = userBuySpent[msg.sender] + buyAmount;
 // ...
}
```

Recommendation(s): Consider accurately updating the `userBuySpent` mapping based on the amount spent by the user: `buyAmount - refund`.

Status: Fixed

Update from the client: addressed in 62ebf712bdc741ed369189f1845877e58b3f23fb

6.5 [Best Practices] Unused variables

File(s): IMO.sol

Description: The IMO contract defines the variables stakes and totalSupplyLimit, but they are never referenced or used within the code.

```
mapping(address => uint256) public stakes;
uint256 public totalSupplyLimit
```

Recommendation(s): Remove the unused variables from the contract to improve code clarity.

Status: Fixed

Update from the client: addressed in 62ebf712bdc741ed369189f1845877e58b3f23fb

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about ORA documentation

Throughout the review process, the `ORA` team offered assistance during meetings to clarify the protocol and address any questions or concerns raised by the Nethermind Security team. However, the reviewed code lacks NatSpec documentation and inline comments explaining the implementation details.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 3 files with 0.8.24
[] Solc 0.8.20 finished in 3.37s
Compiler run successful!
```

8.2 Tests Output

```
> forge test
[] Compiling...
No files changed, compilation skipped

Running 1 test for test/IMO.t.sol:IMOTest
[PASS] test_buyIMOToken() (gas: 97293)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 25.49ms
Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer