Security Review Report
NM-0108 WORLDCOIN ERC20 TOKEN

NETHERMIND

(Jun 27, 2023)

Contents

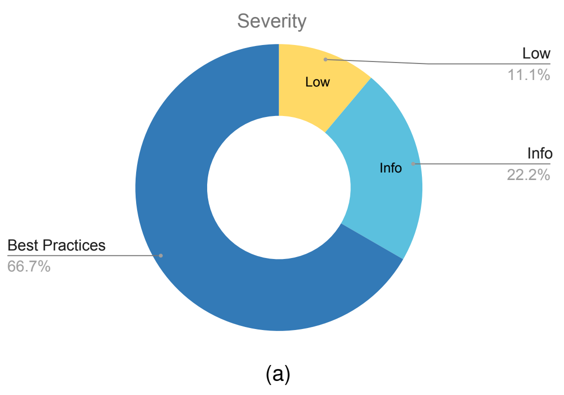
1 Executive Summary

This document presents the security review performed by Nethermind on the World ID ERC20 contract. **World ID** is a privacy-first identity protocol that brings global proof of personhood to the internet. It enables verified users to prove they are unique individuals and, optionally, demonstrate that they perform a given action only once. This is accomplished with a combination of blockchain technology and zero-knowledge proofs (ZKP).

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** 9 points of attention, where 1 is classified as `Low`, 2 are classified as `Informational`, and 6 are classified as `Best Practices`. The issues are summarized in Fig. 1.

During the reaudit, the WorldCoin team has addressed all the identified issues, including the implementation of a modified inflation mechanism, resulting in a reduction in the operational and storage expenses associated with the protocol.

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.



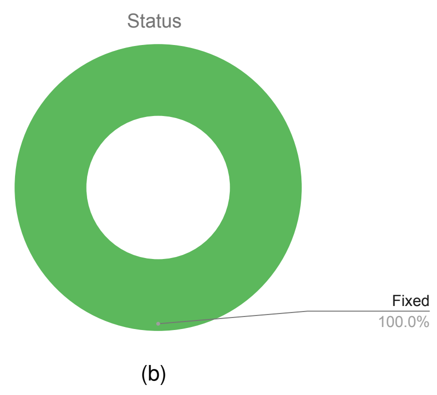


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (2), **Best Practices** (6).
Distribution of status: Fixed (9), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jun. 19, 2023
Response from Client	Jun. 26, 2023
Final Report	Jun. 27, 2023
Methods	Manual Review, Automated Analysis
Repository	https://github.com/worldcoin/worldcoin-token/
Commit Hash (Audit)	37db525451448fa773f91e139facaeedc32041c4
Commit Hash (Reaudit)	9bf99e40f39f4cbf22b177c5f2f8f09f0481c93a cd67a05069aa7d647111ec7da6338db474414efb 3d06ec3bbd06954edab3ae03ce4a118c7fb71546 c8ea6b8c00b6536670f10ae8a0ce6d8fcaa4da4a
Documentation	README.md
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/WLD.sol	110	103	93.6%	43	256
	Total	**110**	**103**	**93.6%**	**43**	**256**

3 Summary of Issues

	Finding	Severity	Update
1	Potential gas-cost griefing due to unbounded `supplyHistory` array	Low	Fixed
2	Function `mint(...)` allows a mint `amount` of zero	Info	Fixed
3	Token decimals can be changed	Info	Fixed
4	Function `_advanceInflationPeriodCursor(...)` can be optimized	Best Practices	Fixed
5	Function `_getConstructionTime(...)` can be optimized	Best Practices	Fixed
6	Functions that can have `external` visibility	Best Practices	Fixed
7	Missing event emission in `constructor(...)`	Best Practices	Fixed
8	Missing input validations in `constructor(...)`	Best Practices	Fixed
9	Storage variables can be immutable	Best Practices	Fixed

4 System Overview

The audit consists of a smart contract for a token called WLD, an ERC20 token. The contract includes various functionalities related to the token, such as minting new tokens, managing ownership, setting metadata, and implementing inflation checks.

The contract begins with several import statements that bring in external libraries and contracts required for its functionality. These imports include the Ownable contract from the OpenZeppelin library, which provides ownership-related functionalities, and the ERC20 contract, which implements the ERC20 standard for token functionality.

The contract itself is defined as WLD and inherits from both the ERC20 contract and the Ownable2Step contract. The Ownable2Step contract is a custom contract that extends the functionality of the Ownable contract, adding an additional step for transferring ownership.

The contract includes a section for storage variables, where various important data related to the token and its inflation are stored. Some of the storage variables include:

minter: This stores the address of the minter, who has the authority to mint new tokens.

```
/// @notice The address of the minter
address public minter;
```

_inflationCapPeriod: This stores the window where inflation cannot be greater than $\frac{\texttt{\_inflationCapNumerator}}{\texttt{\_inflationCapDenominator}}\%$.

```
uint256 private _inflationCapPeriod;
```

_inflationCapNumerator and _inflationCapDenominator: These variables define the inflation cap for the token. The inflation cap restricts the total supply of tokens from increasing by more than a certain percentage within a specific period of time;

```
uint256 private _inflationCapNumerator;
uint256 private _inflationCapDenominator;
```

_inflationPeriodCursor: This stores the last position of the last inflation period inside the array supplyHistory.

```
uint256 private _inflationPeriodCursor;
```

_mintLockInPeriod: This variable indicates the duration when new tokens cannot be minted;

```
/// @notice How many seconds until the mint lock-in period is over
uint256 private _mintLockInPeriod;
```

_symbol, _name, and _decimals: These variables store the symbol, name, and number of decimal places for the token, respectively;

```
/// @notice The symbol of the token
string private _symbol;

/// @notice The name of the token
string private _name;

/// @notice The number of decimals for the WLD token
uint8 private _decimals;
```

supplyHistory: This array stores information about the token's supply at different points in time. Each element in the array contains a timestamp and the corresponding amount of tokens in circulation.

```
struct SupplyInfo {
    uint256 timestamp;
    uint256 amount;
}

/// @notice The history of the WLD token's supply
SupplyInfo[] public supplyHistory;
```

The contract defines several custom error messages using the `error` keyword. These errors are used to revert the contract's execution in case of specific conditions not being met. The defined errors include `CannotRenounceOwnership`, `MintLockInPeriodNotOver`, `NotMinter`, and `InflationCapReached`.

```
/// @notice Emitted in revert if the mint lock-in period is not over.
error MintLockInPeriodNotOver();

/// @notice Emmitted in revert if the caller is not the minter.
error NotMinter();

/// @notice Emmitted in revert if the inflation cap has been reached.
error InflationCapReached();

/// @notice Emmitted in revert if the owner attempts to resign ownership.
error CannotRenounceOwnership();
```

In the constructor function of the contract, various initial parameters are set, including the token's name, symbol, decimals, inflation cap variables, mint lock-in period, initial token holders, and their corresponding amounts. The constructor initializes the inherited `ERC20` contract and the `Ownable` contract with the address of the contract deployer. It also performs some validation checks on the input parameters.

The contract includes several external getter functions that provide information about the token's supply history. These functions allow querying the size of the supply history array, as well as retrieving specific timestamps and amounts from the supply history.

Following the getter functions, there are some metadata-related functions, such as `name()`, `symbol()`, and `decimals()`, which return the name, symbol, and decimal places of the token, respectively. The ERC20 standard requires these functions.

The contract also includes various admin actions that can be performed only by the contract owner. These actions allow the owner to modify the name, symbol, and decimals of the token, set a new minter address, and prevent the owner from renouncing ownership.

Additionally, there are minter actions that can be performed only by the minter address. These actions include the `mint()` function, which mints new tokens and assigns them to a target address. This function implements inflation checks to ensure that the total supply of tokens does not exceed the inflation cap within a specific period. The function updates the supply history with the new information and mints the requested token amount.

The smart contract provides a comprehensive set of functionalities for managing the `WLD` token. It ensures controlled minting of new tokens, tracks the token's supply history, and allows for customization of token metadata. The contract also implements checks to prevent excessive inflation and provides ownership management capabilities.

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 is a measure of how likely the finding is to be uncovered and exploited by an attacker. 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 the finding were to be exploited by an attacker. 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] Potential gas-cost griefing due to unbounded `supplyHistory` array

File(s): `WLD.sol`

Description: The `supplyHistory` array tracks the supply amount at a given timestamp and is used to get the supply amount one `period` ago to check if inflation has exceeded the specified range. Each time the function `mint(...)` is called, a new `SupplyInfo` struct is created and stored in the array. The array is defined as follows.

```
struct SupplyInfo {
    uint256 timestamp;
    uint256 amount;
}
/// @notice The history of the WLD token's supply
SupplyInfo[] public supplyHistory;
```

The function `_advanceInflationPeriodCursor(...)` reads from `supplyHistory` and will loop through the contents from the current cursor until the timestamp is more than one `period` ago. It is possible for a malicious user to repeatedly mint small amounts in a single transaction to greatly increase the entries in `supplyHistory`. When one `period` of time has passed, and the inflation period cursor tries to advance, it will have to loop over these entries. If enough entries have been added, the gas cost required to SLOAD each entry in the `supplyHistory` array may be enough to deter users from minting.

An attacker could combine this with the ability to mint with zero to reduce gas costs while adding entries into `supplyHistory`.

It should be noted that this type of attack would be of a greater cost to the attacker than the victim, from calculations, we estimate it to be a roughly 10x reduction, i.e., it would cost the attacker 1,000,000 gas to cause a cost of 100,000 gas to the victim minter one `period` in the future. However, given that the `period` may be up to one year, changing prices in Ether and gwei-per-gas may change this to be more favorable for the attacker. The function `mint(...)` is shown below.

```
function mint(address to, uint256 amount) public {
    _requireMinter();
    _requirePostMintLockInPeriod();
    _advanceInflationPeriodCursor();
    uint256 oldTotal = _getTotalSupplyInflationPeriodAgo();
    uint256 newTotal = totalSupply() + amount;
    _requireInflationCap(oldTotal, newTotal);
    supplyHistory.push(SupplyInfo(block.timestamp, newTotal));
    _mint(to, amount);
}
```

The function `_advanceInflationPeriodCursor(...)` is reproduced below.

```
function _advanceInflationPeriodCursor() internal {
    uint256 currentTimestamp = block.timestamp;
    uint256 currentPosition = _inflationPeriodCursor;
    // Advancing the cursor until the first of:
    // * we reach the end of the array, or
    // * we reach the first timestamp such that the next timestamp is
    //   younger than _inflationCapPeriod.
    // That means that the cursor will point to the youngest timestamp that
    // is older than said period, i.e. the supply at _inflationCapPeriod ago.
    while (
        currentPosition + 1 < supplyHistory.length &&
        supplyHistory[currentPosition + 1].timestamp + _inflationCapPeriod <
        currentTimestamp
    ) {
        currentPosition += 1;
    }
    _inflationPeriodCursor = currentPosition;
}
```

Recommendation(s): In the function `mint(...)` before pushing to `supplyHistory`, consider checking if the last element in the array has the same timestamp as the current block timestamp. If the timestamps are the same, directly overwrite the supply data in the last element rather than pushing a new entry. This approach prevents single-transaction attacks, allowing an attacker to only inflate the entries in `supplyHistory` once per block, making attack cost non-viable.

Another solution that the WorldCoin team can consider is: the array, along with its associated complexity and costs, can be eliminated by slightly modifying the requirement, as outlined below:

- **Actual Requirement:** Inflation must remain below a given threshold for any time window considered of size `_inflationCapPeriod`;
- **Alleviated Requirement:** Inflation must remain below the given threshold within predefined time windows of `_inflationCapPeriod` seconds;

The **alleviated requirement** introduces logical time blocks (currently set at one year). Within each block, inflation consistently stays within the threshold. As a result, it is only necessary to retain information about the current time block to ensure inflation remains under control.

However, this solution permits the protocol to mint the entire amount of inflation tokens at the end of the current block and mint another amount within the first second of the subsequent block, thereby breaking the invariant established in the **actual requirement**.

Status: Fixed

Update from the client: Fixed in the `commit hash` c8ea6b8c00b6536670f10ae8a0ce6d8fcaa4da4a. Inflation requirements have been alleviated as described here.

6.2 [Info] Function `mint(...)` allows a mint `amount` of zero

File(s): `WLD.sol`

Description: The function `mint(...)` is missing input validation to prevent the argument `amount` from being zero. This will lead to `newTotal` and `oldTotal` holding the same value and then passed to `_requireInflationCap(...)`, the check will always pass (as long as the inflation limit hasn't already been exceeded). On its own, this does not cause any potential issues since no tokens are minted. However, a push to `supplyHistory` is still executed, enabling the above finding to be potentially exploited at a cheaper gas cost to the attacker than if it were impossible to mint zero amounts. The code with audit comments is reproduced below.

```
function mint(address to, uint256 amount) public {
    _requireMinter();
    ////////////////////////////////////////////////////////////////////////////
    // @audit We can mint zero tokens
    ////////////////////////////////////////////////////////////////////////////
    _requirePostMintLockInPeriod();
    _advanceInflationPeriodCursor();
    uint256 oldTotal = _getTotalSupplyInflationPeriodAgo();
    uint256 newTotal = totalSupply() + amount;
    _requireInflationCap(oldTotal, newTotal);
    supplyHistory.push(SupplyInfo(block.timestamp, newTotal));
    _mint(to, amount);
}
```

We note that this function is only controllable by the `minter` address. However, if this address is a smart contract in control of minting, then it may be possible for that external contract to be manipulated into executing zero-amount mints.

Recommendation(s): Consider adding a check to prevent the argument `amount` from being zero.

Status: Fixed

Update from the client: Fixed in the `commit hash` 9bf99e40f39f4cbf22b177c5f2f8f09f0481c93a.

6.3 [Info] Token decimals can be changed

File(s): `WLD.sol`

Description: A token that can change decimals may cause issues if used with other DeFi protocols, such as lending/borrowing. If this token has value, then the decimals should be fixed to ensure compatibility with other DeFi protocols. The function with audit comments is reproduced below.

```
function setDecimals(uint8 decimals_) public onlyOwner {
    _decimals = decimals_;
    //////////////////////////////////////////////////////////////////////
    // @audit Changing the number of decimals can create issues in other
    //        parts of the application
    // @audit Missing event emission
    // @audit external function
    //////////////////////////////////////////////////////////////////////
}
```

Recommendation(s): Consider removing the function `setDecimals(...)` so that it is impossible to change the token decimals after deployment.

Status: Fixed

Update from the client: Fixed in the `commit hash` cd67a05069aa7d647111ec7da6338db474414efb.

6.4 [Best Practices] Function `_advanceInflationPeriodCursor(...)` can be optimized

File(s): `WLD.sol`

Description: The function `_advanceInflationPeriodCursor(...)` can be optimized by not caching the `block.timestamp`, and by caching the length of the `supplyHistory` array. The code, along with audit comments, is provided below:

```
function _advanceInflationPeriodCursor() internal {
    uint256 currentTimestamp = block.timestamp;
    ///////////////////////////////////////////////////////////////////////
    // @audit No need to cache the timestamp because it only costs two gas to
    // add to stack costs more in memory management to cache. Instead,
    // just use `block.timestamp` directly, and should save more gas.
    ///////////////////////////////////////////////////////////////////////
    uint256 currentPosition = _inflationPeriodCursor;
    while (
        ///////////////////////////////////////////////////////////////////
        // @audit You can cache `supplyHistory.length` in memory to save
        // SLOADs every loop
        ///////////////////////////////////////////////////////////////////
        currentPosition + 1 < supplyHistory.length &&
        supplyHistory[currentPosition + 1].timestamp + _inflationCapPeriod < currentTimestamp ) {
        currentPosition += 1;
    }
    _inflationPeriodCursor = currentPosition;
    ///////////////////////////////////////////////////////////////////////
    // @audit Can avoid an unnecessary SLOAD by checking if
    // `currentPosition` is going to be the same as the current cursor,
    // if so, then don't SSTORE. No point in writing to a storage slot the
    // exact same value. This will save SSTORE gas costs when the cursor
    // doesn't need to advance.
    ///////////////////////////////////////////////////////////////////////
}
```

Recommendation(s): This suggestion saves a small amount of gas and is optional.

Status: Fixed

Update from the client: Obsolete after the Commit Hash c8ea6b8c00b6536670f10ae8a0ce6d8fcaa4da4a. The function does not exist anymore.

6.5 [Best Practices] Function `_getConstructionTime(...)` can be optimized

File(s): `WLD.sol`

Description: The function `_getConstructionTime(...)` returns the time the contract was deployed. This time is found by reading from the first entry in the `supplyHistory` array, which requires a `SLOAD`. Since the construction time would not change once deployed, this could be an immutable variable, and no storage would need to be read. The code, along with audit comments, is provided below:

```
function _getConstructionTime() internal view returns (uint256) {
    ///////////////////////////////////////////////////////////////////////////
    // @audit We could have an immutable variable called "constructionTime"
    // That way, you don't have to use a SLOAD every time you want to mint
    // (because `_requirePostmintLockInPeriod()` check)
    // Should save ~2200 gas per mint
    ///////////////////////////////////////////////////////////////////////////
    return supplyHistory[0].timestamp;
}
```

Recommendation(s): Consider storing construction time in an immutable variable rather than reading from storage.

Status: Fixed

Update from the client: Obsolete after the Commit Hash c8ea6b8c00b6536670f10ae8a0ce6d8fcaa4da4a. The function does not exist anymore.

6.6 [Best Practices] Functions that can have `external` visibility

File(s): `WLD.sol`

Description: Some functions are marked as `public`, although they are not called within the contract, and could have a visibility of `external` instead. Using the best-fit visibility can help improve gas costs and code readability, as it clearly separates internal and external logic, making it easier for developers to read the code. A list of functions that can have an `external` visibility are listed below:

```
setName(string memory)
setSymbol(string memory)
setDecimals(uint8)
setMinter(address)
renounceOwnership()
mint(address, uint256)
```

Recommendation(s): Consider changing the visibility of the functions listed above from `public` to `external`.

Status: Fixed

Update from the client: Fixed in the `commit hash` cd67a05069aa7d647111ec7da6338db474414efb.

6.7 [Best Practices] Missing event emission in `constructor(...)`

File(s): `WLD.sol`

Description: It is considered a best practice to emit an event within the `constructor(...)` function, including the input parameters and the contract owner, to facilitate a post-deployment verification of whether the contract has been successfully deployed with the correct parameters.

Recommendation(s): Consider emitting an event within the `constructor(...)` function, including the input parameters and the contract owner.

Status: Fixed

Update from the client: Fixed in

Commit 9bf99e40f39f4cbf22b177c5f2f8f09f0481c93a.

6.8 [Best Practices] Storage variables can be immutable

File(s): `WLD.sol`

Description: The contract contains storage variables that are never changed once set but are not `immutable`. Changing these variables to `immutable` will store their value at constructor time in the contract bytecode rather than storage, saving gas costs since a `SLOAD` does not need to be executed. This also improved code clarity, as it is clear to readers that these variables will not change once set. A list of storage variables that can be set to `immutable` are listed below:

```
uint256 private _inflationCapPeriod;
uint256 private _inflationCapNumerator;
uint256 private _inflationCapDenominator;
uint256 private _mintLockInPeriod;
```

Recommendation(s): Consider setting the storage variables above `immutable`.

Status: Fixed

Update from the client: Fixed in the `commit hash` 3d06ec3bbd06954edab3ae03ce4a118c7fb71546.

6.9 [Best Practices] Missing input validations in `constructor(...)`

File(s): `WLD.sol`

Description: Some arguments passed to the constructor during contract deployment are not validated. Although the constructor is solely invoked during the deployment phase, implementing certain validations could eliminate the need for contract redeployment. The following validations are recommended:

- Consider verifying that `inflationCapDenominator_` is non-zero;
- Consider verifying that `inflationCapPeriod_` is non-zero;
- Consider establishing assumptions regarding the relationship between `inflationCapNumerator_` and `inflationCapDenominator_`;
- Consider defining a valid range for `_mintLockInPeriod_`;

The constructor is reproduced below.

```
constructor(
    string memory name_,
    string memory symbol_,
    uint8 decimals_,
    uint256 inflationCapPeriod_,
    uint256 inflationCapNumerator_,
    uint256 inflationCapDenominator_,
    uint256 mintLockInPeriod_,
    address[] memory initialHolders,
    uint256[] memory initialAmounts
) ERC20(name_, symbol_) Ownable(msg.sender) {
    _name = name_;
    _symbol = symbol_;
    _decimals = decimals_;
    ////////////////////////////////////////////////////////////////////
    // @audit Should we validate "_inflationCapPeriod"?
    ////////////////////////////////////////////////////////////////////
    _inflationCapPeriod = inflationCapPeriod_;

    ////////////////////////////////////////////////////////////////////
    // @audit "_inflationCapDenominator" cannot be zero
    ////////////////////////////////////////////////////////////////////
    _inflationCapNumerator = inflationCapNumerator_;
    _inflationCapDenominator = inflationCapDenominator_;
    ////////////////////////////////////////////////////////////////////
    // @audit "_mintLockInPeriod" not checked for a valid range
    ////////////////////////////////////////////////////////////////////
    _mintLockInPeriod = mintLockInPeriod_;
    _inflationPeriodCursor = 0;
    require(initialAmounts.length == initialHolders.length);
    for (uint256 i = 0; i < initialHolders.length; i++) {
        _update(address(0), initialHolders[i], initialAmounts[i]);
    }
    supplyHistory.push(SupplyInfo(block.timestamp, totalSupply()));
    ////////////////////////////////////////////////////////////////////
    // @audit Missing event emission
    ////////////////////////////////////////////////////////////////////
}
```

Recommendation(s): Consider if the recommendations above align with the primary objectives of the WorldCoin token. These suggestions are optional. However, it is strongly advised to guarantee that the `inflationCapDNumerator_` and `inflationCapDenominator_` are not zero.

Status: Fixed

Update from the client: Added validations for `denominator` and `period`. `Numerator` and `lock period` are intentionally left unconstrained, as every value has a logical and intuitively clear interpretation. Commit 9bf99e40f39f4cbf22b177c5f2f8f09f0481c93a.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. 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 plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. 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 that describes the design and technical details of the smart contract. 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.

Since this is an `ERC20` contract, not much documentation is required to perform the audit.

8 Test Suite Evaluation

```
> forge build
[] Compiling...
[] Compiling 2 files with 0.8.19
[] Solc 0.8.19 finished in 1.39s
Compiler run successful
```

8.1 Tests Output

```
forge test
[] Compiling...
[] Compiling 1 files with 0.8.19
[] Solc 0.8.19 finished in 1.31s
Compiler run successful

Running 11 tests for test/WLD.t.sol:WLDTest
[PASS] testIInitialDistributionHappens() (gas: 16849)
[PASS] testInflationCap() (gas: 323905)
[PASS] testInitialDistributionHappens() (gas: 16805)
[PASS] testInitialDistributionRestricted(address) (runs: 1000, : 10900, ~: 10900)
[PASS] testMintAccessControl(address) (runs: 1000, : 18654, ~: 18654)
[PASS] testMintsLockInPeriod() (gas: 133555)
[PASS] testRenounceOwnershipReverts() (gas: 13442)
[PASS] testSetDecimals(uint8) (runs: 1000, : 19669, ~: 19675)
[PASS] testSetMinterSucceeds(address) (runs: 1000, : 19325, ~: 19325)
[PASS] testSetNameSucceeds(string) (runs: 1000, : 58806, ~: 66659)
[PASS] testSetSymbol(string) (runs: 1000, : 58461, ~: 66659)
Test result: ok. 11 passed; 0 failed; finished in 127.01ms
```

8.2 Code Coverage

```
> forge coverage
```

The relevant output is presented below.

```
] Compiling...
[] Compiling 25 files with 0.8.19
[] Solc 0.8.19 finished in 1.91s
Compiler run successful
Analysing contracts...
Running tests...
| File        | % Lines        | % Statements   | % Branches    | % Funcs        |
|-------------|----------------|----------------|---------------|----------------|
| src/WLD.sol | 90.32% (28/31) | 91.18% (31/34) | 100.00% (6/6) | 77.78% (14/18) |
| Total       | 90.32% (28/31) | 91.18% (31/34) | 100.00% (6/6) | 77.78% (14/18) |
```

8.3 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

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.

Disclaimer