Security Review Report
NM-0112 WORLDCOIN

World ID Contracts - State Bridge Contracts - Airdrop Contracts - Grants Contracts - WLD ERC20 Contract - Vesting Wallet Contract

NETHERMIND

(Jul 23, 2023)

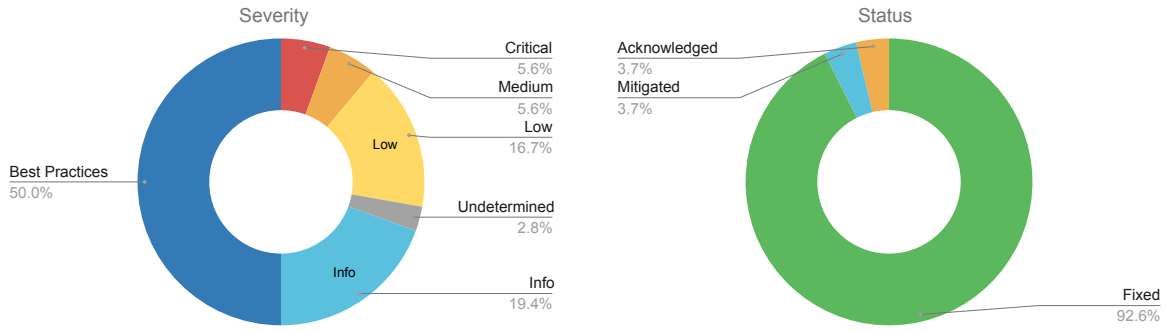
Contents

1 Executive Summary

This document presents the security review performed by Nethermind on:

- World ID contracts containing 700 lines of code with 89.22% of code coverage;
- World ID State Bridge contracts containing 232 lines of code with 32.60% of code coverage; and
- World ID Airdrop contracts containing 123 lines of code with 100% of code coverage;
- WorldCoin Grants Contracts containing 151 lines of code;
- WLD ERC20 Contract containing 110 lines of code;
- Vesting Wallet Contract containing 86 lines of code;
- Audit of the new version of the WLD ERC20 Contract containing 194 lines of code.

The document consolidates all the audits performed by Nethermind from Apr. 2023 to Jul. 2023. The audits were performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, (d) creation of test cases, and (e) use of automated tools. This report provides a comprehensive overview of the audited smart contracts. This includes a detailed technical analysis of each contract, emphasizing data structures, algorithms, flow diagrams, the client's response to each issue, compilation and test output, and code coverage. **Fig. 1(a) presents the issues reported during these audits according to the severity**. We raised 27 points of attention. **Fig. 1(b) presents the final status of the issues after the reaudit stage**. The WorldCoin team has `fixed` 25 issues, `acknowledged` 1 issue, and `mitigated` 1 issue.



(a) distribution of issues

(b) status of issues

Fig. 1: Distribution of issues: Critical (2), **High** (0), **Medium** (2), **Low** (6), **Undetermined** (1), **Informational** (7), **Best Practices** (18). **Distribution of status: Fixed** (25), **Acknowledged** (1), **Mitigated** (1), **Unresolved** (0)

This document is organized as follows. Section 2 presents the files audited by Nethermind. Section 3 summarizes the issues reported by Nethermind for each audit. Section 4 presents the risk rating methodology adopted for the audits. Section 5 presents the audit report for the WorldID, State Bridge, and AirDrop Contracts. Section 6 presents the audit for the WLD ERC20 Contract. Section 7 presents the audit report for the Grants Contracts. Section 8 presents the audit for the Vesting Wallet Contract. Section 9 concludes the document.

Summary of the Audit for World ID, State Bridge, and Airdrop contracts

Audit Type	Security Review
Initial Report	Apr. 17, 2023
Response from Client	May 01, 2023
Final Report	May 22, 2023
Methods	Manual Review, Automated Analysis
Repository	WorldID, State Bridge, Airdrop
Commit Hash - WorldID	20fc82fb52125bf33eb150d45c096441ae516296
Commit Hash - State Bridge	6d01d2178d935c9aa669cf239888f8b1603ee63b
Commit Hash - Airdrop	1150365d420498899efce5d4d5b3f83daa85f98d
Documentation	WorldID Docs WorldId - README State Bridge - README Airdrop - README
Documentation Assessment	High
Test Suite Assessment	High

Summary of the Audit for the WLD ERC20 Contract

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

Summary of the Audit for the Grants Contracts

Audit Type	Security Review
Initial Report	Jul. 11, 2023
Response from Client	Jul. 14, 2023
Final Report	Jul. 20, 2023
Methods	Manual Review, Automated Analysis
Repository	https://github.com/worldcoin/worldcoin-grants-contracts/tree/312294d991c93919ff22a84972d40568506869fa
Commit Hash (Audit)	312294d991c93919ff22a84972d40568506869fa
Documentation	README.md
Documentation Assessment	High
Test Suite Assessment	Medium

Summary of the Audit for the Vesting Wallet Contract

Audit Type	Security Review
Initial Report	Jul. 14, 2023
Response from Client	Jul. 16, 2023
Final Report	Jul. 17, 2023
Methods	Manual Review, Automated Analysis
Repository	https://github.com/worldcoin/worldcoin-token/tree/83ae4a747bd4a7595f8c59f5f5e3d7b66340502b/src
Commit Hash (Audit)	83ae4a747bd4a7595f8c59f5f5e3d7b66340502b
Documentation	README.md
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

2.1 WorldID Contracts

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/WorldIDIdentityManager.sol	6	24	400.0%	4	34
2	src/WorldIDRouter.sol	6	22	366.7%	4	32
3	src/WorldIDRouterImplV1.sol	121	170	140.5%	45	336
4	src/WorldIDIdentityManagerImplV1.sol	431	437	101.4%	104	972
5	src/abstract/WorldIDProxy.sol	6	19	316.7%	3	28
6	src/abstract/WorldIDImpl.sol	14	42	300.0%	6	62
7	src/utils/CheckInitialized.sol	16	17	106.2%	6	39
8	src/utils/UnimplementedTreeVerifier.sol	17	19	111.8%	3	39
9	src/utils/SemaphoreTreeDepthValidator.sol	8	8	100.0%	1	17
10	src/data/VerifierLookupTable.sol	41	74	180.5%	20	135
11	src/interfaces/IWorldID.sol	10	16	160.0%	1	27
12	src/interfaces/IBridge.sol	4	4	100.0%	1	9
13	src/interfaces/IWorldIDGroups.sol	11	18	163.6%	1	30
14	src/interfaces/ITreeVerifier.sol	9	16	177.8%	1	26
	Total	**700**	**886**	**126.6%**	**200**	**1786**

2.2 State Bridge Contracts

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/StateBridge.sol	74	72	97.3%	32	178
2	src/OpWorldID.sol	16	39	243.8%	8	63
3	src/PolygonWorldID.sol	28	40	142.9%	10	78
4	src/abstract/WorldIDBridge.sol	76	109	143.4%	39	224
5	src/utils/SemaphoreTreeDepthValidator.sol	8	7	87.5%	1	16
6	src/interfaces/ICrossDomainOwnable3.sol	4	9	225.0%	1	14
7	src/interfaces/IWorldID.sol	10	17	170.0%	1	28
8	src/interfaces/IWorldIDIdentityManager.sol	4	9	225.0%	1	14
9	src/interfaces/IBridge.sol	4	7	175.0%	1	12
10	src/interfaces/ISendBridge.sol	4	7	175.0%	1	12
11	src/interfaces/IOpWorldID.sol	4	10	250.0%	1	15
	Total	**232**	**326**	**140.5%**	**96**	**654**

2.3 Airdrop Contracts

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/WorldIDAirdrop.sol	56	53	94.6%	26	135
2	src/WorldIDMultiAirdrop.sol	67	61	91.0%	31	159
	Total	**123**	**114**	**92.7%**	**57**	**294**

2.4 WLD ERC20 Contract

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

2.5 Grants Contracts

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/HourlyGrant.sol	19	4	21.1%	7	30
2	src/WeeklyGrant.sol	19	1	5.3%	6	26
3	src/IGrant.sol	7	8	114.3%	4	19
4	src/MonthlyGrant.sol	35	14	40.0%	7	56
5	src/RecurringGrantDrop.sol	71	61	85.9%	32	164
	Total	**151**	**88**	**58.3%**	**56**	**295**

2.6 Vesting Wallet Contracts

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/WLD.sol	106	88	83.0%	37	231
2	src/VestingWallet.sol	40	37	92.5%	9	86
	Total	**146**	**125**	**85.6%**	**46**	**317**

3 Summary of Issues

3.1 WorldID, State Bridge, and AirDrop Contracts

	Finding	Severity	Update
1	Airdrops' funds can be stolen from the `WorldIDMultiAirdrop` contract	Critical	Fixed
2	Old roots can allow invalid proofs to be submitted as valid in the contract `WorldIdBridge`	Critical	Fixed
3	Centralization Risks	Medium	Mitigated
4	`PolygonWorldID` sender and receiver can be set by any address	Medium	Fixed
5	Absence of `ROOT_HISTORY_EXPIRY` update mechanism in `StateBridge` contract	Low	Fixed
6	Lack of a two-step process for transferring ownership	Low	Fixed
7	`NonExistentRoot()` may not be executed for nonexistent roots	Low	Fixed
8	`actionId` is hashed two times in `WorldIDAirdrop` contract	Low	Fixed
9	`ActionID` may not be unique across applications	Undetermined	Acknowledged
10	Double initialization of the tree depth	Info	Fixed
11	Unnecessary storage writes	Info	Fixed
12	Lack of events for relevant operations	Best Practices	Fixed
13	Low-level calls may cause errors	Best Practices	Fixed
14	Non-private functions are not virtual	Best Practices	Fixed
15	Presence of unused variables	Best Practices	Fixed

3.2 WLD ERC20 Contract

	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

3.3 Grants Contracts

	Finding	Severity	Update
1	Solmate's `safeTransferFrom(...)` may silently fail if the specified token has no code	Low	Fixed
2	Missing input validation in `MonthlyGrant.constructor(...)`	Info	Fixed
3	Possible underflow when retrieving current `grantId`	Info	Fixed
4	Ensure that you are the owner after deployment the contract `RecurringGrantDrop`	Info	Fixed
5	Confusing date description in `MonthlyGrant.getCurrentId()`	Best Practices	Fixed
6	Defining Storage Variables after Events and Errors	Best Practices	Fixed
7	Function that can have `external` visibility	Best Practices	Fixed
8	Missing event during sensitive storage variable update	Best Practices	Fixed
9	Use of Solidity's date and time units instead of calculations	Best Practices	Fixed
10	`RecurringGrantDrop.addAllowedCaller(...)` not checking `_caller` for `address(0x0)`	Best Practices	Fixed
11	Function `checkClaim(...)` may have `internal` visibility	Best Practices	Fixed
12	Functions `addAllowedCaller(...)` and `removeAllowedCaller(...)` not emitting events	Best Practices	Fixed

4 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;

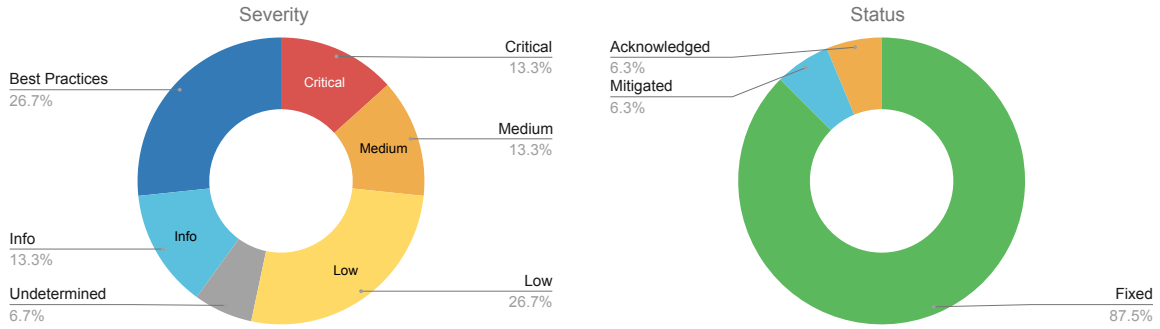
c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

5 WorldID & State Bridge & AirDrop Contracts Audit Report

This section presents the security review performed by Nethermind on the World ID contracts. 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 Blockchain technology and zero-knowledge proofs (ZKP). The audited code consists of three different components listed below:

- World ID contracts containing 700 lines of code with 89.22% of code coverage;
- World ID State Bridge contracts containing 232 lines of code with 32.60% of code coverage; and
- World ID Example Airdrop contracts containing 123 lines of code with 100% of code coverage.

Each component includes a `README` file briefly introducing the code and instructions for using, testing, and deploying these contracts. Moreover, official documentation was utilized during this assessment. The review was supported by open communication with the development team, allowing for a more comprehensive understanding of the protocol mechanisms and addressing any potential gaps. Along this section, we report 15 points of attention, where two are classified as `Critical`, two are classified as `Medium`, four are classified as `Low`, six are classified as `Informational` or `Best Practices`, and one is classified as `Undetermined`. The issues are summarized in Fig. 2.



(a) distribution of issues

(b) status of issues

Fig. 2: Distribution of issues: Critical (2), **High** (0), **Medium** (2), **Low** (4), **Undetermined** (1), **Informational** (2), **Best Practices** (4). **Distribution of status: Fixed** (13), **Acknowledged** (1), **Mitigated** (1), **Unresolved** (0)

5.1 Technical Overview: World ID Contracts & World ID State Bridge

The audit encompasses three components: a) `World ID contracts`, comprising fourteen contracts; b) `World ID state bridge`, consisting of eleven contracts; and c) `World ID airdrop`, containing two contracts. Fig. 3 illustrates a structural diagram of the contracts and their relationships. Subsequently, we outline the primary contracts (highlighted in yellow): `VerifierLookupTable`, `WorldIDIdentityManager`, `WorldIDRouter`, `StateBridge`, `PolygonWorldID`, and `OpWorldID`.

VerifierLookupTable: This contract establishes a mapping between batch sizes and tree verifiers, returning the suitable verifier for a specific batch size before employing that verifier to validate a tree modification proof. The contract inherits from the `Ownable` contract, which offers a straightforward access control mechanism, ensuring only the contract owner can modify the lookup table. The contract features several public functions for interacting with the lookup table:

a. The `getVerifierFor(...)` function takes a batch size as input and returns the associated tree verifier instance. It checks that the batch size is valid, i.e., that there is an associated verifier for that size.

b. The `addVerifier(...)` function allows the owner to add a new verifier for a given batch size. It checks that there is no existing verifier for the given batch size and returns an error if there is.

c. The `updateVerifier(...)` function allows the owner to update an existing verifier for a given batch size.

d. The `disableVerifier(...)` function allows the owner to disable a verifier for a given batch size by setting it to null.

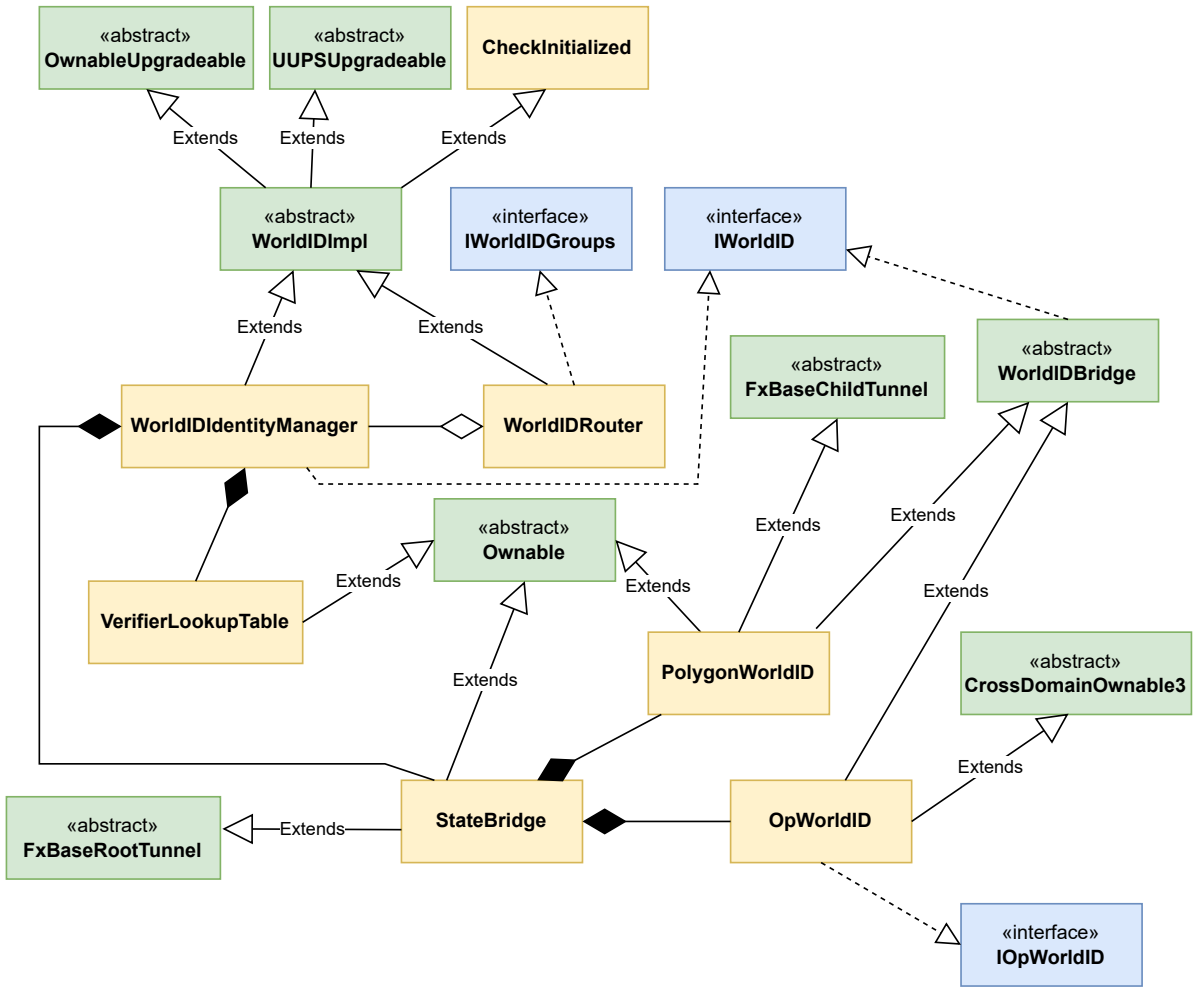


Fig. 3: Class diagram of the main contracts and their relationships.

WorldIDIdentityManager: Designed to operate behind a proxy, this contract implements a batch-based identity manager for the WorldID protocol, which verifies externally created Zero Knowledge Proofs. The contract features various public functions serving different purposes, such as:

a. The functions `setRegisterIdentitiesVerifierLookupTable(...)` and `setIdentityUpdateVerifierLookupTable(...)`, which allow setting lookup tables of verifiers used for identity updates.

b. The function `setSemaphoreVerifier(...)` enables setting the address for the semaphore verifier used for verifying semaphore proofs.

c. The functions `enableStateBridge(...)` and `disableStateBridge(...)`, which enable or disable the state bridge.

d. The function `verifyProof(...)`, which allows verifying semaphore proofs.

e. The `updateIdentities(...)` and `registerIdentities(...)` functions, allowing the owner to modify the current registry of identities in a group. It is worth mentioning that most checks are done when modifying the identities in the group are executed in the proof generation phase and not performed on the smart contracts layer.

The contract owner can only call all of the previously mentioned functions, except for the `verifyProof(...)`. Two of the functions, `registerIdentities(...)` and `updateIdentities(...)`, require meeting specific conditions for successful verification. After meeting these conditions, the latest root is updated, the previous root is added to history, and its expiration timestamp is set. Fig. 4 shows a sequence diagram to illustrate how contracts interact with each other to register identities. As can be observed, `registerIdentities(...)` sends the latest WorldID Identity Manager root to all chains.

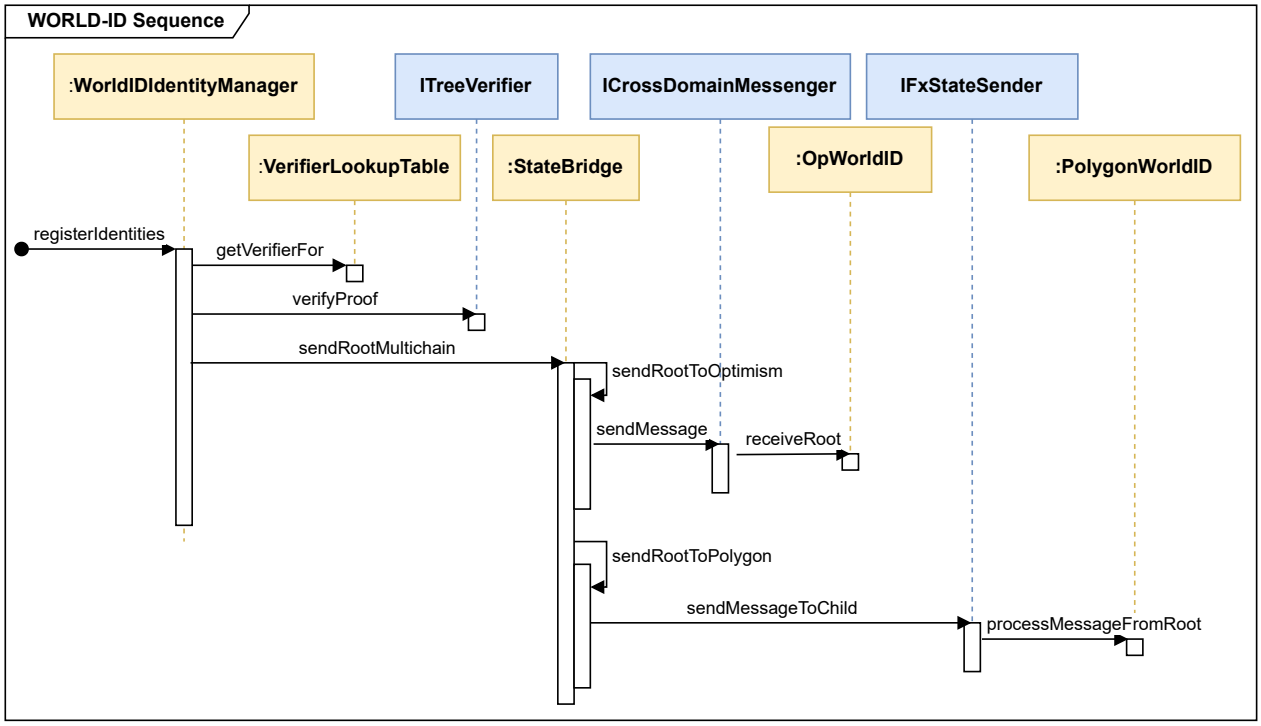


Fig. 4: Sequence diagram for the function `WorldIDIdentityManager.registerIdentities(...)`.

WorldIDRouter: This contract is designed to operate behind a proxy. Specifically, `WorldIDRouter` is responsible for dispatching group numbers to the correct identity manager implementation. The contract implements several public functions that serve different purposes:

a. `routeFor(...)` returns the target address to specific groups based on a provided group number.

b. `addGroup(...)` adds a new group to the router.

c. `updateGroup(...)` updates the target address for a group in the router.

d. `disableGroup(...)` disables the target group in the router.

e. `groupCount(...)` returns the number of groups in the routing table.

StateBridge: The `StateBridge` contract is responsible for distributing new World ID Identity Manager roots and timestamps to the supported chains, namely Polygon and Optimism. This contract lives on Ethereum Mainnet and is called by the `WorldIDIdentityManager` contract through the `registerIdentities(...)` and `updateIdentities(...)` functions described below:

a. `sendRootMultichain(...)` is used to transmit the latest WorldID Identity Manager root to all chains supported by the WorldID Identity Manager contract. The root is provided as an argument to this function. Before transmitting, this function verifies that the caller is the WorldID Identity Manager contract. Fig. 4 describes the sequence to send the root to the chains.

b. `transferOwnershipOptimism(...)` allows the `StateBridge` contract to transfer ownership of `OpWorldID` to another contract on L1 or to a local Optimism EOA. This function can only be called by the contract owner.

PolygonWorldID: The `PolygonWorldID` contract manages the root history of the WorldID Merkle root on Polygon PoS. Its primary goal is to allow semaphore-proof verifications on the Polygon PoS chain. As described in Fig. 3, it inherits from the `WorldIDBridge`, `FxBaseChildTunnel`, and `Ownable` contracts. This contract is deployed on Polygon PoS and is invoked by the `StateBridge` contract for each new root insertion. Fig. 4 illustrates the internal function `StateBridge.sendRootToPolygon` sending the new root to `PolygonWorldID`. This contract implements the following functions:

a. `setRootHistoryExpiry(...)` sets the amount of time it takes for a root in the root history to expire. It takes in one parameter: `expiryTime`, which is the new amount of time it takes for a root to expire. This function can only be called by the contract owner.

b. `_processMessageFromRoot(...)` is an internal override function called via a system call to receive messages from the `StateBridge` contract. It calls the `_receiveRoot` function upon receiving a message from the `StateBridge` contract via the `FxChildTunnel`.

c. The function `verifyProof(...)`, which allows verifying semaphore proofs.

OpWorldID: Similarly to the `PolygonWorldID` contract, `OpWorldID` manages the root history of the WorldID Merkle root, but it is deployed on the Optimism chain. Its primary goal is to allow semaphore-proof verifications on the Optimism chain. As described in Fig. 3, it inherits from the `WorldIDBridge` and `CrossDomainOwnable3` contracts and implements the `IOpWorldID` interface. This contract is invoked by the `StateBridge` contract for each new root insertion. Fig. 4 illustrates the internal function `StateBridge.sendRootToOptimism` sending the new root to `OpWorldID`. This contract implements the following functions:

a. `receiveRoot(...)` is called by the state bridge contract when it forwards a new root to the bridged WorldID. It sets the new root and supersedes the timestamp in the root history.

b. `setRootHistoryExpiry(...)` sets the expiration time for a root in the root history to expire. Only the owner of the contract can call this function.

c. The function `verifyProof(...)`, which allows verifying semaphore proofs.

5.2 World ID AirDrop

This component consists of two template contracts for WorldID users: a) `WorldIDAirdrop` is aimed for airdropping tokens to World ID members, and b) `WorldIDMultiAirdrop` manages multiple airdrops to World ID users. Fig. 5 illustrates the interactions between contracts in the sequential order to claim the airdrop in `WorldIDAirdrop` contract.

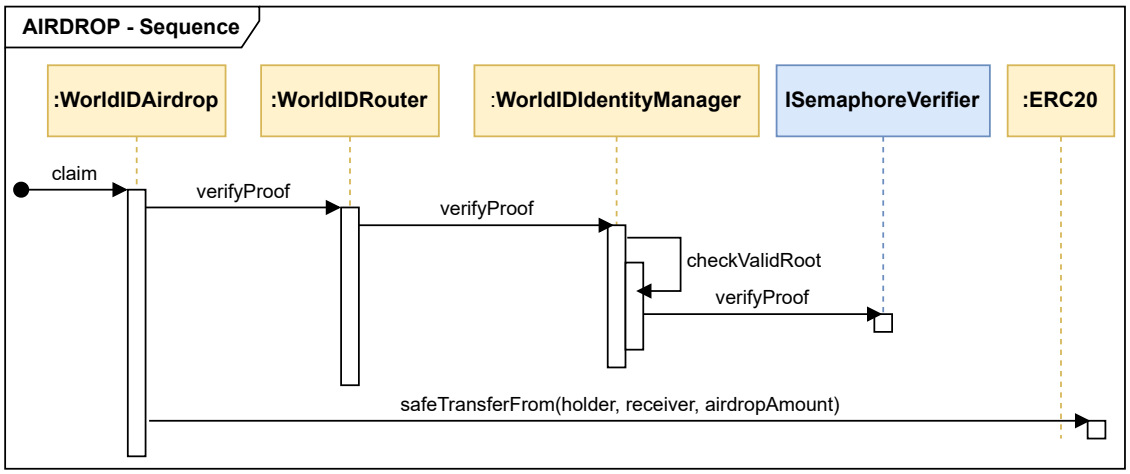


Fig. 5: Sequence diagram for the function `WorldIDAirdrop.claim(...)`.

5.3 Abstract Formal Specification for Poseidon hash

In this section, we present a formal proof of equivalence between implementing the Poseidon hash and a zero-knowledge circuit for it using the Lean proof assistant. Firstly, we provide a brief overview of the Poseidon hash (Section 5.3.1), followed by a discussion of the Lean formalization in Section 5.3.2.

5.3.1 An overview of Poseidon hash

Let $\mathbb{F}_p$ be a prime field where p is approximately greater than 2^{31}, i.e., $p \approx 2^n > 2^{31}$. The Poseidon hash is a function that maps strings from $\mathbb{F}p$ to fixed-length strings over $\mathbb{F}p^o$, where o is usually equal to 1.Implementing the Poseidon hash uses the concept of *rounds*, executed over the state a number of times. Each round applies an `S-box` function to the entire state (full rounds) or a single value, while the rest remains unchanged (half-rounds). Intuitively, each round function consists of the following three steps:

a. A step for adding constants, which sums a vector to a column of the state representation;

b. A subwords step applies an `S-box` function to the state. When the `S-box` is applied to just one state entry; we call this a half-round. Applying `S-box` to all state entries is named a full round.

c. A mixing step multiplies the current state by an MDS matrix.

The next Section presents more details about the Lean formalization of Poseidon hash and the proof of equivalence with a zero-knowledge circuit using the Lean proof assistant.

5.3.2 Lean formalization

Here, we present more details about the Poseidon hash implementation in Lean and proof of its equivalence with encoding a `ZK-circuit` as a relational specification. For each implementation component, we show its Lean implementation and its corresponding `ZK-circuit` together with a theorem statement establishing its correctness.

Definition of a finite field type: We start our formalization by defining a type for denoting finite fields:

```
def Order : ℕ := -- some prime number
variable [Fact (Nat.Prime Order)]
abbrev F := ZMod Order
```

We start by defining a constant `Order` for denoting the order of our finite field type. We postulate the fact that `Order` is a prime number and then define a type `F` as the type of integers modulus the prime `Order`.

Function sbox: The first function defined for the Poseidon implementation is `sbox`:

```
def sbox(v : F) : F := v ^ 5
```

The function takes a field value and raises it to the 5th power. The following proposition describes the circuit definition for this function:

```
def sboxConstraint (v : F)(res : F) : Prop :=
  ∃ v₂ v₄, v₂ = v * v ∧ v₄ = v₂ * v₂ ∧ res = v * v₄
```

Finally, we can say that sbox function is sound by stating the following theorem:

```
theorem sboxSound (v res : F) : sboxConstraint v res ↔ sbox v = res
```

The theorem ensures that sbox result agrees with the circuit defined by `sboxConstraint`.

Function applyMDS: The next function definition is `applyMDS`, which multiplies the current state by an MDS matrix. Its implementation uses Lean Mathlib's matrix multiplication operation.

```
  def applyMDS {n : ℕ} (state : Matrix (Fin (n + 1)) Unit F)
                       (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F) : Matrix (Fin (n + 1)) Unit F
        := Matrix.mul mds state
```

The circuit for expressing matrix multiplication is defined by quantifying all possible rows of the output as follows:

```
def applyMDSConstraint
    {n : ℕ}
    (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (state : Matrix (Fin (n + 1)) Unit F)
    (res : Matrix (Fin (n + 1)) Unit F) : Prop :=
        ∀ i, res i unit = Σ v, (mds i v) * state v unit
```

The soundness theorem for the function `applyMDS` ensures that the function's result agrees with the circuit implementation:

```
theorem applyMDSSound {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                              (state : Matrix (Fin (n + 1)) Unit F)
                              (res : Matrix (Fin (n + 1)) Unit F) :
      applyMDSConstraint mds state res ↔ applyMDS state mds = res
```

The Lean proof script for this theorem can be found here.

Function half-round: The definition of the function `half-round` takes the current state, the MDS matrix, the index of the column that should be used by the actual round, the matrix of constants, and it performs the following steps:

- Get the column that the current round should operate on;
- Add the previously obtained column to the current state value;
- Apply the sbox function to the first entry of the new state value;
- Apply the MDS matrix to the newly updated state.

```
def halfRound {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                     (pround : Fin (n + 1))
                     (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                     (state : Matrix (Fin (n + 1)) Unit F)
            : Matrix (Fin (n + 1)) Unit F :=
        let constm := col (constants pround)
        let state' := state + constm
        let state'' : Matrix (Fin (n + 1)) Unit F := updateEntry state'
      applyMDS state'' mds
```

The relational definition of the `ZK-circuit` for `half-round` is:

```
def halfRoundConstraint {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (pround : Fin (n + 1))
                               (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (state : Matrix (Fin (n + 1)) Unit F)
                              (res : Matrix (Fin (n + 1)) Unit F) : Prop :=
        ∃ res₁ res₂,
        sumGadget state (col (constants pround)) res₁ ∧
        updateEntryConstraint res₁ res₂ ∧
        applyMDSConstraint mds res₂ res
```

it uses the `sumGadget`, the constraint representation for the sum of two vectors. The definition of the `sumGadget` constraint can be found here. The soundness theorem for the function `half-round` ensures that the function's result agrees with its circuit implementation:

```
theorem halfRoundSound {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                              (pround : Fin (n + 1))
                               (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (state : Matrix (Fin (n + 1)) Unit F)
                               (res : Matrix (Fin (n + 1)) Unit F) :
      halfRoundConstraint mds pround constants state res ↔ halfRound mds pround constants state = res
```

The complete Lean proof script for the soundness theorem can be found online here.

Function full-round: The definition of the function `full-round` takes the current state, the MDS matrix, the index of the column that should be used by the actual round, and the matrix of constants, and it performs the following steps:

- Get the column that the current round should operate on;
- Add the previously obtained column to the current state value;
- Apply the sbox function all entries of the current state;
- Apply the MDS matrix to the newly updated state.

```
def fullRound {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (fround : Fin (n + 1))
                        (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (state : Matrix (Fin (n + 1)) Unit F)
            : Matrix (Fin (n + 1)) Unit F :=
        let constm := col (constants fround)
        let state' := state + constm
        let state'' := Matrix.map state' sbox
        applyMDS state'' mds
```

The relational definition of the `ZK-circuit` for `full-round` is:

```
def fullRoundConstraint  {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                                 (pround : Fin (n + 1))
                                 (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                                 (state : Matrix (Fin (n + 1)) Unit F)
                                 (res : Matrix (Fin (n + 1)) Unit F) : Prop :=
          ∃ res₁, sumGadget state (col (constants pround)) res₁ ∧
          ∃ res₂, (∀ i, sboxConstraint (res₁ i unit) (res₂ i unit)) ∧
          applyMDSConstraint mds res₂ res
```

The soundness theorem for the function `full-round` ensures that the function's result agrees with its circuit implementation:

```
theorem fullRoundSound {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (pround : Fin (n + 1))
                               (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (state : Matrix (Fin (n + 1)) Unit F)
                               (res : Matrix (Fin (n + 1)) Unit F) :
      fullRoundConstraint mds pround constants state res ↔ fullRound mds pround constants state = res
```

The complete Lean proof script for the soundness theorem can be found online here.

Function poseidonIter: The function `poseidonIter` implements the core functionality of the Poseidon hashing by iterating `nTotal` times over the state. Based on the current iteration, the algorithm decides if it should apply a half or full round to the current state. After `nTotal` iterations the function `poseidonIter` returns the resulting state matrix.

```
def poseidonIter {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (nTotal nFull : ℕ)
                        (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (state : Matrix (Fin (n + 1)) Unit F)
      : ∀ (i : ℕ), Matrix (Fin (n + 1)) Unit F :=
      λ i =>
        if h : i < nTotal + 1 then
           let state' := if i < nFull || i < (nTotal - nFull) then
                                      fullRound mds i constants state
                                  else halfRound mds i constants state
           poseidonIter mds nTotal nFull constants  state' (i + 1)
        else state
termination_by poseidonIter mds nTotal nFull constants state i => nTotal + 1 - i
decreasing_by {apply minus_lt ; exact h}
```

The clauses `termination_by` and `decreasing_by` are used by the Lean equation compiler to guarantee the `poseidonIter` does terminate. The `ZK-circuit` for `poseidonIter` is defined as:

```
def poseidonIterConstraint {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                                  (nTotal nFull : ℕ)
                                  (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F) :
                                  Matrix (Fin (n + 1)) Unit F →
                                  Matrix (Fin (n + 1)) Unit F →  ℕ → Prop :=
     λ state res i =>
         if h : i < nTotal + 1 then
           if i < nFull ∨ i < (nTotal - nFull) then
                ∃ res₁, fullRoundConstraint mds i constants state res₁ ∧
                poseidonIterConstraint mds nTotal nFull constants res₁ res (i + 1)
           else
                ∃ res₁, halfRoundConstraint mds i constants state res₁ ∧
                poseidonIterConstraint mds  nTotal nFull constants res₁ res (i + 1)
         else res = state
termination_by poseidonIterConstraint _ nTotal _ _ _ _ i => nTotal + 1 - i
decreasing_by {apply minus_lt ; exact h}
```

The soundness proof for `poseidonIterConstraint` ensures that the functions result agrees with its circuit implementation:

```
theorem poseidonIterSound  {n : ℕ}
                           (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                           (nTotal nFull : ℕ)
                           (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F) :
                           ∀ state res i,
        poseidonIterConstraint mds nTotal nFull constants state res i ↔
        poseidonIter mds nTotal nFull constants state i = res :=
```

The complete proof script for `poseidonIterSound` can be found at here.

Function poseidon: The function `poseidon` implements the hashing algorithm by getting the updated state by `poseidonIter` and returns the first entry of the resulting matrix.

```
def poseidon {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                    (nTotal nFull : ℕ)
                    (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                    (state : Matrix (Fin (n + 1)) Unit F) : F :=
      let state' := poseidonIter mds nTotal nFull constants state 0
      state' 0 unit
```

The relational definition of the `ZK-circuit` for `poseidon` is:

```
def poseidonConstraint
    {n : ℕ}
    (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (nTotal nFull : ℕ)
    (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (state : Matrix (Fin (n + 1)) Unit F)(res : F) : Prop :=
    ∃ res₁, poseidonIterConstraint mds nTotal nFull constants state res₁ 0 ∧
                    res = res₁ 0 unit
```

The soundness theorem for the function `poseidon` ensures that the function's result agrees with its circuit implementation:

```
theorem poseidonSound
    {n : ℕ}
    (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (nTotal nFull : ℕ)
    (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (state : Matrix (Fin (n + 1)) Unit F)
    (res : F) :
      poseidonConstraint mds nTotal nFull constants state res ↔
      poseidon mds nTotal nFull constants state = res
```

The complete Lean proof script for the soundness theorem for `poseidon` can be found online here.

5.3.3 Final Remarks

We presented an abstract formal specification for the Poseidon hash function as `ZK-circuits` using the Lean proof assistant. The formalization is composed of three main components: 1) a specification of the Poseidon hash algorithm as Lean functions; 2) the implementation of Poseidon hash as `ZK-circuits` implemented as Lean propositions over prime fields and 3) proofs ensuring the soundness of the circuit representations with respect to the Poseidon hash specification. The complete Lean formalization is available at https://github.com/NethermindEth/WorldCoin-poseidon-verification.

5.4 Findings: WorldID & State Bridge & AirDrop Contracts

5.4.1 [Critical] Airdrops' funds can be stolen from the `WorldIDMultiAirdrop` contract

File(s): `WorldIDMultiAirdrop.sol`

Description: The `WorldIDMultiAidrop` contract allows any user to create a new airdrop by calling the `createAirdrop(...)` function:

```
function createAirdrop(uint256 groupId, ERC20 token, address holder, uint256 amount) public {
    Airdrop memory airdrop = Airdrop({
        groupId: groupId,
        token: token,
        manager: msg.sender,
        ////////////////////////////////////////////////////////////////////////////
        // @audit "holder" argument can be any address
        ////////////////////////////////////////////////////////////////////////////
        holder: holder,
        amount: amount
    });

    getAirdrop[nextAirdropId] = airdrop;
    emit AirdropCreated(nextAirdropId, airdrop);

    ++nextAirdropId;
}
```

To create an airdrop, the creator needs to provide four parameters:

- `groupId`: The WorldID group containing all eligible recipients for the airdrop;
- `token`: The asset to be airdropped;
- `holder`: The account holding the funds to be airdropped. This account must approve sufficient funds for the `WorldIDMultiAirdrop`;
- `amount`: The amount sent to each user claiming the airdrop;

When a user claims their reward for a specific airdrop, they provide the airdrop's `id` and data proving their membership in the eligible group. If the proof is valid, the `WorldIDMultiAirdrop` contract transfers the reward from the airdrop's `holder` to the `receiver`.

```
worldIdRouter.verifyProof(
    root,
    airdrop.groupId,
    abi.encodePacked(receiver).hashToField(),
    nullifierHash,
    abi.encodePacked(address(this), airdropId).hashToField(),
    proof
);

nullifierHashes[nullifierHash] = true;
emit AirdropClaimed(airdropId, receiver);
////////////////////////////////////////////////////////////////////////////
// @audit - Assets are sent from the "holder" to the "receiver" parameter
////////////////////////////////////////////////////////////////////////////
SafeTransferLib.safeTransferFrom(airdrop.token, airdrop.holder, receiver, airdrop.amount);
```

As there are no restrictions on the `holder` parameter during airdrop creation, a malicious user can create a new airdrop using any address that has granted an allowance to the `WorldIDMultiAirdrop` contract as the `holder`. The attacker can then claim the airdrop, effectively stealing assets from the `holder`'s account.

Recommendation(s): Create separate contracts for each airdrop to ensure that the `holder` only grants allowance to the specific airdrop.

Status: Fixed.

Update from client: We decided that we won't use WorldIDMultiAirdrop.sol in production anymore and we have deleted the contract from the repo. Fixed in commit world-id-example-airdrop@b830bf8.

5.4.2 [Critical] Old roots can allow invalid proofs to be submitted as valid in the contract `WorldIdBridge`

File(s): WorldIDBridge.sol

Description: The `WorldIdBridge.verifyProof(...)` function enables verification of the proof provider's group membership. As groups are subject to change over time, the validity of roots endures for some time after their replacement. To verify the validity of a given root at the present time, the `checkValidRoot(...)` function is employed. This function is intended to return true for valid roots and to revert for invalid ones. The function with audit comments is reproduced below.

```
function checkValidRoot(...) public view returns (bool isValid) {
    if (root == _latestRoot) {
        return true;
    }

    uint128 rootTimestamp = rootHistory[root];

    if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
       revert ExpiredRoot();
    }

    if (rootTimestamp == 0) {
       revert NonExistentRoot();
    }
    ////////////////////////////////////////////////////
    // @audit When the old block is still valid
    //        the value of `isValid` is false
    ////////////////////////////////////////////////////
}
```

The function `verifyProof(...)` is expected to revert when invalid proofs are provided. The function is shown in the next code snippet.

```
function verifyProof(...) public view virtual onlyProxy onlyInitialized {
    ////////////////////////////////////////////
    // @audit when `checkValidRoot` returns false
    //        the proof is not verified by
    //        `semaphoreVerifier`
    ////////////////////////////////////////////
    if (checkValidRoot(root)) {
        semaphoreVerifier.verifyProof( root, nullifierHash, signalHash, externalNullifierHash, proof, treeDepth );
    }
}
```

Nevertheless, it is evident from the aforementioned code that the `checkValidRoot(...)` function produces a `false` return value for older, albeit still valid, roots, which is an erroneous outcome. This particular flaw carries a greater risk, given that the `verifyProof(...)` function does not execute the `verifyProof(...)` function from the semaphore contract in instances where the outcome of `checkValidRoot(...)` is `false`. As a result, `verifyProof(...)` will not revert even in the case of an invalid proof, which may enable an attacker to submit invalid proofs as valid when an old root remains valid.

Recommendation(s): Add the `return` statement in the `checkValidRoot(...)` function with the proper value when the `root` is still valid.

Status: Fixed.

Update from the client: Fixed in commits `world-id-state-bridge@5a340f6`, world-id-state-bridge@1ad9dc2

5.4.3 [Medium] Centralization Risks

Description: The reviewed contracts contain multiple sensitive actions that can only be executed by accounts with special roles. These actions include:

- Upgrading the logic executed by the system;
- Managing the roots representing identities in each group;
- Modifying the verifier contracts utilized for semaphore and group change proofs;
- Handling data sent to bridges;
- Altering expiration times for roots;
- Managing the routing table for accessing the appropriate WorldID contract;
- Managing contracts' ownership;

These actions could render the system unusable or damage its users. In the event of private key leaks or similar incidents, malicious actors could easily harm the protocol and its users.

Recommendation(s): To mitigate this issue, consider implementing the following measures:

- Utilize a multi-signature wallet as the holder of any privileged role;
- Implement a time lock for any sensitive action, allowing users and developers to intervene in case of malicious intent to harm the protocol;
- Provide clear and comprehensive documentation on any privileged actions within the protocol and the roles authorized to execute them;

Status: Mitigated.

Update from the client: Fixed in commit `world-id-contracts@dd7b7f6`.

5.4.4 [Medium] `PolygonWorldID` sender and receiver can be set by any address

File(s): `PolygonWorldID.sol`, `StateBridge.sol`

Description: The `PolygonWorldID` contract is responsible for managing the root history of the `WorldID` Merkle root on Polygon PoS. Each new root insertion is called by `StateBridge` (`L1`). The mechanism that is used for sending the state from Ethereum to Polygon PoS consists of two actors: Child and Root. The `PolygonWorldID` is the Child since it's a message receiver, and the `StateBridge` is a Root that sends a state. Both contracts inherit Child and Root functionality from FxBaseChildTunnel and FxBaseRootTunnel, respectively. The issue is that functions for setting Child address in Root contract, and Root address in Child contract are `public`/ `external`. This creates a situation where a transaction for setting the `Child` and the `Root` may be front-run by an adversarial actor. Below we present those functions:

FxBaseChildTunnel.setFxRootTunnel(...)

```
function setFxRootTunnel(address _fxRootTunnel) external virtual {
    require(fxRootTunnel == address(0x0), "FxBaseChildTunnel: ROOT_TUNNEL_ALREADY_SET");
    fxRootTunnel = _fxRootTunnel;
}
```

FxBaseRootTunnel.setFxChildTunnel(...)

```
function setFxChildTunnel(address _fxChildTunnel) public virtual {
    require(fxChildTunnel == address(0x0), "FxBaseRootTunnel: CHILD_TUNNEL_ALREADY_SET");
    fxChildTunnel = _fxChildTunnel;
}
```

Recommendation(s): There are at least two approaches for solving this issue: a) set the Child/Root address in the constructor when deploying contracts. Addresses of deployed contracts can be known upfront; b) overwrite the setting functions and add the `onlyOwner` modifier to prevent front-running the call.

Status: Fixed.

Update from the client: Fixed in commit `world-id-state-bridge@5310dfa`.

5.4.5 [Low] Absence of `ROOT_HISTORY_EXPIRY` update mechanism in `StateBridge` contract

File(s): `OpWorldID.sol`

Description: The `OpWorldID` contract uses the `onlyOwner` modifier for the functions `receiveRoot(...)` and `setRootHistoryExpiry(...)`, as illustrated below.

```
contract OpWorldID is WorldIDBridge, CrossDomainOwnable3, IOpWorldID {
    ...
    function receiveRoot(...) external virtual onlyOwner {
        _receiveRoot(newRoot, supersedeTimestamp);
    }

    ////////////////////////////////////////////////////////////
    // @audit Only the StateBridge contract can call this function
    ////////////////////////////////////////////////////////////
    function setRootHistoryExpiry(...) public virtual override onlyOwner {
        _setRootHistoryExpiry(expiryTime);
    }
}
```

The `receiveRoot(...)` function is called by the `StateBridge` contract (the owner) to forward the latest `root` to the bridged WorldID. In contrast, `setRootHistoryExpiry(...)` updates the expiration time for a `root`. However, the `StateBridge` contract does not have a function for calling `setRootHistoryExpiry(...)` to update the expiration time. To invoke this function, the owner of `StateBridge` must transfer ownership of `OpWorldID` to another contract or a local Optimism EOA.

Recommendation(s): Add an external `onlyOwner` function in the `StateBridge` to call `OpWorldID.setRootHistoryExpiry(...)`.

Status: Fixed.

Update from the client: Fixed in commit `world-id-state-bridge@ee74770`.

5.4.6 [Low] Lack of a two-step process for transferring ownership

File(s): `WorldIDImpl.sol`

Description: The contract `WorldIDImpl` is derived from OpenZeppelin's `OwnableUpgradeable` contract, which furnishes a single-step ownership transfer. The `transferOwnership(...)` function facilitates the transfer of ownership of the contract in one step. However, if the contract's owner is assigned to an address that is not managed by the Worldcoin team, reclamation of ownership of the contract will not be feasible.

Failing to use such a mechanism can introduce several issues. If the ownership transfer is not properly controlled and executed, it can create security vulnerabilities in the contract. Without proper access control, a malicious actor could gain control of the contract and perform unauthorized actions.

Changing ownership without proper access controls and protocols can lead to a lack of transparency and accountability in the transfer process. This can harm the trust and credibility of the contract and its stakeholders.

Recommendation(s): Consider implementing a two-step process for transferring ownership, such as the propose-accept scheme. Check the Ownable2StepUpgradeable.sol contract from OpenZeppelin.

Status: Fixed.

Update from the client: Fixed in commit `world-id-contracts@c8a10af`.

5.4.7 [Low] `NonExistentRoot()` may not be executed for nonexistent roots

File(s): `WorldIDBridge.sol`, `WorldIDIdentityManagerImplV1.sol`

Description: The function `checkValidRoot(...)` checks if a given root value is valid. When the `root` is not the latest, the next `if` statement is executed. In case the `root` does not exist, the `rootTimestamp` value is zero, and the function reverts with the error `ExpiredRoot()`, and the condition `if (rootTimestamp == 0)` will not be reached.

```
function checkValidRoot(uint256 root) public view returns (bool isValid) {
    if (root == _latestRoot) { return true; }
    uint128 rootTimestamp = rootHistory[root];
    if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) { revert ExpiredRoot(); }
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // @audit The "if" condition below may not be reached when "block.timestamp>ROOT_HISTORY_EXPIRY" and rootTimestamp=0
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    if (rootTimestamp == 0) { revert NonExistentRoot(); }
}
```

Recommendation(s): Switch the order of the `if` statements as shown below.

```
-// Expired roots are not valid.
-if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
-   revert ExpiredRoot();
-}
-
-// And roots do not exist if they don't have an associated timestamp.
-if (rootTimestamp == 0) {
-   revert NonExistentRoot();
-}
+ // And roots do not exist if they don't have an associated timestamp.
+ if (rootTimestamp == 0) {
+     revert NonExistentRoot();
+ }
+ // Expired roots are not valid.
+ if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
+     revert ExpiredRoot();
+ }
```

Status: Fixed.

Update from the client: Fixed in commits `world-id-contracts@95691df`, `world-id-state-bridge@5a340f6`.

5.4.8 [Low] `actionId` is hashed two times in `WorldIDAirdrop` contract

File(s): `WorldIDAirdrop.sol`

Description: The `WorldIDAirdrop` constructor requires several parameters, as outlined below.

```
constructor(IWorldIDGroups _worldIdRouter, uint256 _groupId, string memory _actionId,
            ERC20 _token, address _holder, uint256 _airdropAmount) {
  worldIdRouter = _worldIdRouter;
  groupId = _groupId;
  actionId = abi.encodePacked(_actionId).hashToField();
  token = _token;
  holder = _holder;
  airdropAmount = _airdropAmount;
}
```

The constructor accepts the `_actionId` as registered in the developer portal, creates a `keccak256` hash of it, and stores the hash in the `actionId` state variable. When a user claims their reward, they provide the data proving their membership in the group, as demonstrated in the function below.

```
function claim(address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof) public
{
  if (nullifierHashes[nullifierHash]) revert InvalidNullifier();
  worldIdRouter.verifyProof(
      groupId,
      root,
      abi.encodePacked(receiver).hashToField(), // The signal of the proof
      nullifierHash,
      ////////////////////////////////////////////////////////////////
      // @audit the "actionId""" is already hashed
      //        in the constructor
      ////////////////////////////////////////////////////////////////
    abi.encodePacked(actionId).hashToField(), // The external nullifier hash
      proof
  );
...
}
```

However, the `claim(...)` function hashes the already hashed `actionId` once more. If the proof is not generated for the double-hashed `actionId`, the `worldIdRouter.verifyProof` will revert.

Recommendation(s): Remove the double hashing.

```
function claim(address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof) public
{
  if (nullifierHashes[nullifierHash]) revert InvalidNullifier();
  worldIdRouter.verifyProof(
      groupId,
      root,
      abi.encodePacked(receiver).hashToField(), // The signal of the proof
      nullifierHash,
      ////////////////////////////////////////
      // @audit the actionId is already hashed
      //        in the constructor
      ////////////////////////////////////////
-         abi.encodePacked(actionId).hashToField(), // The external nullifier hash
+         actionId, // The external nullifier hash
      proof
  );
...
}
```

Status: Fixed.

Update from the client: Fixed in commit world-id-example-airdrop@f7b0cee.

5.4.9 [Undetermined] `ActionID` may not be unique across applications

Description: When verifying a claim through `WorldIDGroups`, a user must provide six elements:

- `groupId`: A group to which the user belongs;
- `root`: The root of the Merkle tree containing all identities in the group;
- `signalHash`: The hash of a message the user commits to;
- `nullifierHash`: An identifier for the proof, derived from the user's identity and `ActionId`;
- `externalNullifierHash`: The `ActionId` represents the user's desired action;
- `proof`: Eight values used to validate the provided arguments;

Provided all the values are correct, the caller proves that:

- The user is part of the group identified by `groupId`;
- The user wants to perform the action represented by `ActionId/ExternalNullifierHash`;
- The user commits to the information represented by `signalHash` for their action;

Although `WorldIDGroup` does not prevent reusing proofs, applications built on top of it can do so using the `nullifierHash`. Allowing a specific `nullifierHash` to be used only once means each user can execute the action identified by `externalNullifierHash` only once.

However, since this check is performed in each application, proofs can still be reused across different applications. This can be mitigated by using unique `ActionIDs` for each application. The `WorldCoin` team recommends building `ActionIDs` using the application address as a prefix, but there are no restrictions preventing a malicious user from using the same `ActionIDs` as other applications.

An attacker could create a phishing application and incentivize users to submit proofs to it, then reuse these proofs in other applications using `WorldID`. The severity and likelihood of this attack vary across applications.

Recommendation(s): In order to mitigate this issue, we recommend doing the following:

- Encourage applications using `WorldID` to follow a well-defined standard for creating `ActionIds`. The `ActionIds` should include identifiers unique to the application and easily verifiable by users, such as the address of the contract;
- Develop tools that allow users to inspect a human-readable version of the `ActionIds`. For example, in addition to the `ActionId`, display the values used to generate it, which could include application identifiers such as the address;
- Encourage users to carefully review the `ActionIds` they are signaling when submitting proofs;

Status: Acknowledged

Update from the client: Working on documentation to explain how this will be/is handled.

5.4.10 [Info] Double initialization of the tree depth

File(s): PolygonWorldID.sol

Description: The constructor of the PolygonWorldID contract checks and sets the tree depth, as we can see in the code below.

```
contract PolygonWorldID is WorldIDBridge, FxBaseChildTunnel, Ownable {

    constructor(uint8 _treeDepth, address _fxChild)
        WorldIDBridge(_treeDepth)
        FxBaseChildTunnel(_fxChild)
    {
        if (!SemaphoreTreeDepthValidator.validate(_treeDepth)) {
            revert UnsupportedTreeDepth(_treeDepth);
        }
        ////////////////////////////////////
        // @audit tree depth is set here //
        ////////////////////////////////////
        treeDepth = _treeDepth;
    }
```

However, the identical process occurs within the WorldIDBridge contract, from which the PolygonWorldID contract inherits. This results in a superfluous assignment of the tree depth.

```
abstract contract WorldIDBridge is IWorldID {
    ...
    constructor(uint8 _treeDepth) {
        if (!SemaphoreTreeDepthValidator.validate(_treeDepth)) {
            revert UnsupportedTreeDepth(_treeDepth);
        }
        ////////////////////////////////////////
        // @audit tree depth is also set here //
        ////////////////////////////////////////
        treeDepth = _treeDepth;
    }
}
```

Recommendation(s): Avoid redundant operations.

Status: Fixed.

Update from the client: Fixed in commit world-id-state-bridge@25e6d85.

5.4.11 [Info] Unnecessary storage writes

File(s): WorldIDRouterImplV1.sol

Description: The `WorldIDRouter` contract utilizes an array as a `routingTable`, which stores all registered groups and identifies them by their index in the array. The array is initialized with a length of `DEFAULT_ROUTING_TABLE_SIZE`. When a new group is added, and the current number of elements in the `routingTable` equals its size, a new array is created in memory with a length equal to the current size plus `DEFAULT_ROUTING_TABLE_GROWTH`. The newly created array is then copied element by element to the storage.

```
function insertNewTableEntry(uint256 groupId, address targetAddress)
    internal
    onlyProxy
    onlyInitialized
{
    while (groupId >= routingTable.length) {
        uint256 existingTableLength = routingTable.length;
        address[] memory newRoutingTable =
            new address[](existingTableLength + DEFAULT_ROUTING_TABLE_GROWTH);

        for (uint256 i = 0; i < existingTableLength; ++i) {
            newRoutingTable[i] = routingTable[i];
        }
        // @audit - This operation overwrites storage values, causing
        //          unnecessary gas consumption.
        routingTable = newRoutingTable;
    }

    routingTable[groupId] = targetAddress;
    _groupCount++;
}
```

This operation overwrites existing storage values with the same values, leading to unnecessary gas consumption.

Recommendation(s): Consider removing reallocation logic and using the `push` method for storage arrays instead.

```
- /// The default size of the internal routing table.
- uint256 internal constant DEFAULT_ROUTING_TABLE_SIZE = 10;

- /// How much the routing table grows when it runs out of space.
- uint256 internal constant DEFAULT_ROUTING_TABLE_GROWTH = 5;

/// The null address.
address internal constant NULL_ADDRESS = address(0x0);

/// The routing table is used to dispatch from groups to addresses.
address[] internal routingTable;

- /// The number of groups currently set in the routing table.
- uint256 internal _groupCount;
```

```
function initialize(address initialGroupIdentityManager) public reinitializer(1) {
    // Initialize the sub-contracts.
    __delegateInit();

    // Now we can perform our own internal initialisation.
-   routingTable = new address[](DEFAULT_ROUTING_TABLE_SIZE);
-   routingTable[0] = initialGroupIdentityManager;
-   _groupCount = 1;
+   routingTable.push(initialGroupIdentityManager);

    // Mark the contract as initialized.
    __setInitialized();
}
```

```
function groupCount() public view onlyProxy onlyInitialized returns (uint256 count) {
-   return _groupCount;
+   return routingTable.length;
}
```

```
function insertNewTableEntry(uint256 groupId, address targetAddress)
    internal
    onlyProxy
    onlyInitialized
{
-    while (groupId >= routingTable.length) {
-        uint256 existingTableLength = routingTable.length;
-        address[] memory newRoutingTable =
-            new address[](existingTableLength + DEFAULT_ROUTING_TABLE_GROWTH);
-
-        for (uint256 i = 0; i < existingTableLength; ++i) {
-            newRoutingTable[i] = routingTable[i];
-        }
-
-        routingTable = newRoutingTable;
-    }
-
-    routingTable[groupId] = targetAddress;
-    _groupCount++;
+    routingTable.push(targetAddress);
}
```

```
function nextGroupId() public view onlyProxy onlyInitialized returns (uint256 count) {
-   return _groupCount;
+   return routingTable.length;
}
```

Status: Fixed.

Update from the client: Fixed in commits `world-id-contracts@95691df`, world-id-contracts@38a5d1c.

5.4.12 [Best Practice] Lack of events for relevant operations

File(s): `WorldIDIdentityManagerImplV1.sol`, `WorldIDRouterImplV1.sol`, `WorldIDBridge.sol`, `WorldIDAirdrop.sol`

Description: Several relevant operations in the contracts do not emit events, making it difficult to monitor and review the contracts' behavior once deployed.

Operations that would benefit from emitting events include:

```
- WorldIDIdentityManagerImplV1.initialize(...)
- WorldIDIdentityManagerImplV1.registerIdentities(...)
- WorldIDIdentityManagerImplV1.updateIdentities(...)
- WorldIDIdentityManagerImplV1.setStateBridge(...)
- WorldIDIdentityManagerImplV1.setSemaphoreVerifier(...)
- WorldIDIdentityManagerImplV1.setRootHistoryExpiry(...)
- WorldIDRouterImplV1.initialize(...)
- WorldIDRouterImplV1.addGroup(...)
- WorldIDRouterImplV1.updateGroup(...)
- WorldIDBridge._setRootHistoryExpiry(...)
- WorldIDAirdrop.claim(...)
```

Emitting events for the relevant operations will enable users and blockchain monitoring systems to easily detect suspicious behaviors and ensure the correct functioning of the contracts.

Recommendation(s): Add events for all relevant operations to facilitate contract monitoring and detect suspicious behavior more effectively.

Status: Fixed.

Update from the client: Fixed in commits `world-id-contracts@5f0f56c`, world-id-state-bridge@aab1ea1, world-id-example-airdrop@f7d60fe, world-id-contracts@d29d6a9.

5.4.13 [Best Practice] Low-level calls may cause errors

File(s): WorldIDRouterImplV1.sol

Description:

The `verifyProof(...)` function, present within the `WorldIDRouter` contract, retrieves the appropriate `IdentityManager` for the indicated group and invokes the `verifyProof(...)` function of that `IdentityManager`. This call to the `IdentityManager` is accomplished by employing the following code.

```
bytes memory callData = abi.encodeCall(
    IWorldID.verifyProof, (root, signalHash, nullifierHash, externalNullifierHash, proof)
);

(bool success,) = identityManager.call(callData);
if (!success) {
    revert FailedToVerifyProof();
}
```

As demonstrated in the code excerpt provided, a low-level `call` instruction is utilized. Utilizing low-level calls is not advisable as they do not ordinarily implement security precautions. For instance, in this particular scenario, if the address pointed to by `identityManager` lacks any code, this function will succeed when it should actually revert.

Recommendation(s): Avoid using low-level calls. Use `IWorldID` as the type of `identityManager`.

Status: Fixed.

Update from the client: Fixed in commit world-id-contracts@95691df.

5.4.14 [Best Practice] Non-private functions are not virtual

File(s): WorldIDRouterImplV1.sol

Description: The optimal practices for implementing contracts are defined at the commencement of `WorldIDRouterImplV1` contract. Nevertheless, the advice to make all non-private functions virtual has not been strictly adhered to, as there exist non-private functions that lack virtual property. This might result in an inability to update these functions if a new implementation were to inherit from the present one.

Recommendation(s): Define non-private functions as virtual in implementation contracts.

Status: Fixed.

Update from the client: Fixed in commits world-id-contracts@95691df, world-id-contracts@5034fec.

5.4.15 [Best Practice] Presence of unused variables

File(s): StateBridge.sol, WorldIDBridge.sol

Description: The presence of unused variables is generally considered a practice to be avoided, as they may lead to potential issues such as:

- Increased computational costs (i.e., unnecessary gas consumption);
- Reduced code readability; and;
- The possibility of bugs (e.g., when an explicit return statement is forgotten and return values are never assigned);

The state variable `worldID` in the `StateBridge` contract is only set in the constructor and is not used throughout the contract.

```

contract StateBridge is FxBaseRootTunnel, Ownable {
...
    // @audit unused state variable
    IWorldIDIdentityManager internal worldID;
...
}

```

Another example of an unused variable is present in the `WorldIDBridge` contract, declared in the function `checkValidRoot(...)`. The function contains the local variable `isValid`, which is never assigned a value, so it assumes its initialization value. When the `if` condition is not satisfied, `false` will always be returned without any warning of the forgotten statement `return`.

```
abstract contract WorldIDBridge is IWorldID  {
...
    function checkValidRoot(uint256 root) public view returns (bool isValid) {
        // @audit a value is never assigned to isValid
        if (root == _latestRoot) {
            return true;
        }

        uint128 rootTimestamp = rootHistory[root];

        // Expired roots are not valid.
        if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
            revert ExpiredRoot();
        }

        // And roots do not exist if they don't have an associated timestamp.
        if (rootTimestamp == 0) {
            revert NonExistentRoot();
        }
    }
...
}
```

List of unused local variables in `return` statements:

```
- WorldIDRouterImplV1
  - function routeFor(...)

- WorldIDBridge
   - function checkValidRoot
   - function latestRoot()
   - function rootHistoryExpiry()
   - function getTreeDepth()

- WorldIDIdentityManagerImplV1
   - function queryRoot(...)
   - function latestRoot() stateBridge()
   - function isInputInReducedForm(...)
   - function reduceInputElementInSnarkScalarField(...)
   - function getRegisterIdentitiesVerifierLookupTableAddress()
   - function getIdentityUpdateVerifierLookupTableAddress()
   - function getSemaphoreVerifierAddress()
   - function getRootHistoryExpiry()
   - function getTreeDepth()
```

Unused state variable:

```
StateBridge
  - IWorldIDIdentityManager internal worldID
```

Recommendation(s): We recommend removing all unused variables from the codebase to enhance code quality and minimize the risk of unexpected errors or inefficiencies.

Status: Fixed.

Update from the client: Fixed in commits `world-id-state-bridge@93fc506`, world-id-state-bridge@1f8366c, world-id-contracts@275e2aa.

5.5 Documentation Evaluation: WorldID & State Bridge & AirDrop Contracts

Software documentation refers to 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.

To assist the audit process, the World ID team provided Unit Tests, Worldcoin docs on the Github repository, and extensive documentation available here. The documentation embraces detailed specifications for the contracts under auditing, e.g., how World ID proofs are verified on-chain and how they use Zero-Knowledge proofs. Notably, their code is rich in inline comments, significantly contributing to understanding the implementation and functionality.

In addition to the official documentation, we also rely on documentation for:

- Semaphore;
- Bridge system used for sending messages to Polygon;
- Bridge system used for sending messages to Optimism.

The documentation provided by the WorldCoin team and the additional sources we utilized enabled us to thoroughly cover the terms used in the source code, core business logic, and function flow. Reviewing the entire documentation suite, we comprehensively understood how the contracts should function. The provided documentation is well-written and structured, making it an excellent resource for developers and auditors. The codebase contains an adequate number of inline comments and well-named functions and variables, which greatly assisted the audit team in understanding the function flow and identifying any potential issues.

5.6 Test Suite Evaluation: WorldID & State Bridge & AirDrop Contracts

5.6.1 Compilation Output: WorldID Contracts

```
> make build
forge build
[] Compiling...
[] Compiling 75 files with 0.8.19
[] Solc 0.8.19 finished in 42.89s
Compiler run successful
```

5.6.2 Compilation Output: State Bridge Contracts

```
> make build
forge build
[] Compiling...
[] Compiling 126 files with 0.8.15
[] Solc 0.8.15 finished in 15.36s
Compiler run successful (with warnings)
warning[6321]: Warning: Unnamed return variable can remain unassigned. Add an explicit return with value to all
 ↪  non-reverting code paths or name the variable.
  --> lib/contracts/contracts/lib/MerklePatriciaProof.sol:20:30:
   |
20 |     ) internal pure returns (bool) {
   |                              ^^^^



warning[5667]: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> src/mock/WorldIDIdentityManagerMock.sol:25:29:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |                             ^^^^^^^^^^^^



warning[2018]: Warning: Function state mutability can be restricted to pure
  --> src/mock/WorldIDIdentityManagerMock.sol:25:5:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |     ^ (Relevant source part starts here and spans across multiple lines).
```

5.6.3 Compilation Output: AirDrop Contracts

```
> make build
forge build
[] Compiling...
[] Compiling 14 files with 0.8.19
[] Solc 0.8.19 finished in 2.06s
Compiler run successful
```

5.6.4 Tests Output: WorldID

```
> make test
FOUNDRY_PROFILE=debug forge test
[] Compiling...
[] Compiling 27 files with 0.8.19
[] Solc 0.8.19 finished in 44.35s
Compiler run successful

Running 1 test for src/test/verifier-lookup-table/VerifierLookupTableConstruction.t.sol:BatchLookupTableConstruction
[PASS] testCanConstructLookupTable() (gas: 268426)
Test result: ok. 1 passed; 0 failed; finished in 1.74ms

Running 2 tests for src/test/router/WorldIDRouterDataQuery.t.sol:WorldIDRouterDataQuery
[PASS] testCanGetGroupCount() (gas: 24168)
[PASS] testCannotGetGroupCountUnlessViaProxy() (gas: 8555)
Test result: ok. 2 passed; 0 failed; finished in 6.90ms

Running 5 tests for
↪  src/test/verifier-lookup-table/VerifierLookupTableOwnershipManagement.t.sol:BatchLookupTableOwnershipManagement
[PASS] testCannotRenounceOwnershipIfNotOwner(address) (runs: 256, : 15830, ~: 15830)
[PASS] testCannotTransferOwnerIfNotOwner(address,address) (runs: 256, : 16119, ~: 16119)
[PASS] testHasOwner() (gas: 9869)
[PASS] testRenounceOwnership() (gas: 10530)
[PASS] testTransferOwner(address) (runs: 256, : 22699, ~: 22699)
Test result: ok. 5 passed; 0 failed; finished in 67.67ms

Running 9 tests for src/test/verifier-lookup-table/VerifierLookupTableQuery.t.sol.sol:VerifierLookupTableQuery
[PASS] testCanAddVerifierWithValidBatchSize(uint256,address) (runs: 256, : 39450, ~: 39450)
[PASS] testCanDisableVerifier(uint256) (runs: 256, : 28202, ~: 28225)
[PASS] testCanGetVerifierForExtantBatchSize() (gas: 12307)
[PASS] testCanUpdateVerifierWithValidBatchSize(address) (runs: 256, : 23916, ~: 23916)
[PASS] testCannotAddVerifierForBatchSizeThatAlreadyExists(uint256) (runs: 256, : 39342, ~: 39342)
[PASS] testCannotAddVerifierUnlessOwner(address) (runs: 256, : 18123, ~: 18123)
[PASS] testCannotDisableVerifierUnlessOwner(address) (runs: 256, : 18089, ~: 18089)
[PASS] testCannotGetVerifierForMissingBatchSize(uint256) (runs: 256, : 13596, ~: 13596)
[PASS] testCannotUpdateVerifierIfNotOwner(address) (runs: 256, : 20353, ~: 20353)
Test result: ok. 9 passed; 0 failed; finished in 225.43ms

Running 5 tests for src/test/router/WorldIDRouterOwnershipManagement.t.sol:WorldIDRouterOwnershipManagement
[PASS] testCannotRenounceOwnershipIfNotOwner(address) (runs: 256, : 48701, ~: 48711)
[PASS] testCannotTransferOwnerIfNotOwner(address,address) (runs: 256, : 49135, ~: 49135)
[PASS] testHasOwner() (gas: 21821)
[PASS] testRenounceOwnership() (gas: 25248)
[PASS] testTransferOwner(address) (runs: 256, : 37763, ~: 37763)
Test result: ok. 5 passed; 0 failed; finished in 385.85ms

Running 15 tests for src/test/identity-manager/WorldIDIdentityManagerUninit.t.sol:WorldIDIdentityManagerUninit
[PASS] testShouldNotCallCalculateIdentityRegistrationInputHash() (gas: 3167241)
[PASS] testShouldNotCallCheckValidRootWhileUninit() (gas: 3157481)
[PASS] testShouldNotCallGetIdentityUpdateVerifierLookupTableAddressWhileUninit() (gas: 3157309)
[PASS] testShouldNotCallGetRootHistoryExpiryWhileUninit() (gas: 3157199)
[PASS] testShouldNotCallGetSemaphoreVerifierAddressWhileUninit() (gas: 3157285)
[PASS] testShouldNotCallIsInputInReducedFormWhileUninit() (gas: 3157414)
[PASS] testShouldNotCallLatestRootWhileUninit() (gas: 3157308)
[PASS] testShouldNotCallQueryRootWhileUninit() (gas: 3157583)
[PASS] testShouldNotCallRegisterIdentitiesWhileUninit() (gas: 3185451)
[PASS] testShouldNotCallSetIdentityUpdateVerifierLookupTableWhileUninit() (gas: 3951369)
[PASS] testShouldNotCallSetRegisterIdentitiesVerifierLookupTableWhileUninit() (gas: 3951414)
[PASS] testShouldNotCallSetRootHistoryExpiryWhileUninit() (gas: 3157465)
[PASS] testShouldNotCallSetSemaphoreVerifierWhileUninit() (gas: 9349980)
[PASS] testShouldNotCallUpdateIdentitiesWhileUninit(uint128[],uint128[8]) (runs: 256, : 3359014, ~: 3367924)
[PASS] testShouldNotCallgetRegisterIdentitiesVerifierLookupTableAddressWhileUninit() (gas: 3157309)
Test result: ok. 15 passed; 0 failed; finished in 1.18s
```

```
Running 7 tests for src/test/identity-manager/WorldIDIdentityManagerCalculation.t.sol:WorldIDIdentityManagerCalculation
[PASS] testCalculateIdentityRegistrationInputHashFromParametersOnKnownInput() (gas: 32679)
[PASS] testCanCalculateIdentityUpdateInputHash(uint256,uint256,uint32,uint32,uint256,uint256,uint256,uint256) (runs:
↪  256, : 27471, ~: 27471)

[PASS] testCanCheckValueIsInReducedForm(uint256) (runs: 256, : 25369, ~: 25369)
[PASS] testCanCheckValueIsNotInReducedForm(uint256) (runs: 256, : 25398, ~: 25398)
[PASS] testCannotCalculateIdentityRegistrationInputHashIfNotViaProxy() (gas: 17895)
[PASS] testCannotCalculateIdentityUpdateHashIfNotViaProxy(uint256,uint256,uint32[],uint256[],uint256[]) (runs: 256, :
↪  79263, ~: 81179)
[PASS] testCannotCheckValidIsInReducedFormIfNotViaProxy() (gas: 8702)
Test result: ok. 7 passed; 0 failed; finished in 1.22s

Running 2 tests for
↪  src/test/identity-manager/WorldIDIdentityManagerConstruction.t.sol:WorldIDIdentityManagerConstruction
[PASS] testCanConstructIdentityManagerWithDelegate() (gas: 3345748)
[PASS] testCanConstructIdentityManagerWithNoDelegate() (gas: 97722)
Test result: ok. 2 passed; 0 failed; finished in 10.11ms

Running 2 tests for src/test/router/WorldIDRouterConstruction.t.sol:WorldIDRouterConstruction
[PASS] testCanConstructRouterWithDelegate(address) (runs: 256, : 1687484, ~: 1687484)
[PASS] testCanConstructRouterWithNoDelegate() (gas: 97679)
Test result: ok. 2 passed; 0 failed; finished in 158.51ms

Running 4 tests for src/test/identity-manager/WorldIDIdentityManagerUpgrade.t.sol:WorldIDIdentityManagerUpdate
[PASS] testCanUpgradeImplementationWithCall() (gas: 3138170)
[PASS] testCanUpgradeImplementationWithoutCall() (gas: 3124384)
[PASS] testCannotUpgradeUnlessManager(address) (runs: 256, : 3149466, ~: 3149466)
[PASS] testCannotUpgradeWithoutProxy() (gas: 3125858)
Test result: ok. 4 passed; 0 failed; finished in 463.88ms

Running 4 tests for
↪  src/test/identity-manager/WorldIDIdentityManagerInitialization.t.sol:WorldIDIdentityManagerInitialization
[PASS] testCannotInitializeTheDelegate() (gas: 3056958)
[PASS] testCannotPassUnsupportedTreeDepth() (gas: 3176393)
[PASS] testInitialisation() (gas: 3351225)
[PASS] testInitializationOnlyOnce() (gas: 62063)
Test result: ok. 4 passed; 0 failed; finished in 23.98ms

Running 21 tests for
↪  src/test/identity-manager/WorldIDIdentityManagerGettersSetters.t.sol:WorldIDIdentityManagerGettersSetters
[PASS] testCanGetIdentityUpdateVerifierLookupTableAddress() (gas: 26479)
[PASS] testCanGetRegisterIdentitiesVerifierLookupTableAddress() (gas: 26488)
[PASS] testCanGetRootHistoryExpiry() (gas: 24286)
[PASS] testCanGetSemaphoreVerifierAddress() (gas: 15236)
[PASS] testCanSetIdentityUpdateVerifierLookupTable() (gas: 825574)
[PASS] testCanSetRegisterIdentitiesVerifierLookupTable() (gas: 825554)
[PASS] testCanSetRootHistoryExpiry(uint256) (runs: 256, : 34474, ~: 34474)
[PASS] testCanSetSemaphoreVerifier() (gas: 6225791)
[PASS] testCannotGetIdentityUpdateVerifierLookupTableAddressUnlessViaProxy() (gas: 8709)
[PASS] testCannotGetRegisterIdentitiesVerifierLookupTableAddressUnlessViaProxy() (gas: 8707)
[PASS] testCannotGetRootHistoryExpiryUnlessViaProxy() (gas: 8635)
[PASS] testCannotGetSemaphoreVerifierAddressUnlessViaProxy() (gas: 8664)
[PASS] testCannotSetIdentityUpdateVerifierLookupTableUnlessOwner(address) (runs: 256, : 841169, ~: 841169)
[PASS] testCannotSetIdentityUpdateVerifierLookupTableUnlessViaProxy() (gas: 802699)
[PASS] testCannotSetRegisterIdentitiesVerifierLookupTableUnlessOwner(address) (runs: 256, : 841147, ~: 841147)
[PASS] testCannotSetRegisterIdentitiesVerifierLookupTableUnlessViaProxy() (gas: 802702)
[PASS] testCannotSetRootHistoryExpiryToZero() (gas: 43650)
[PASS] testCannotSetRootHistoryExpiryUnlessOwner(address) (runs: 256, : 6241419, ~: 6241419)
[PASS] testCannotSetRootHistoryExpiryUnlessViaProxy(uint256) (runs: 256, : 9145, ~: 9145)
[PASS] testCannotSetSemaphoreVerifierAddressUnlessViaProxy() (gas: 6202971)
[PASS] testCannotSetSemaphoreVerifierUnlessOwner(address) (runs: 256, : 6241375, ~: 6241375)
Test result: ok. 21 passed; 0 failed; finished in 1.69s

Running 5 tests for
↪  src/test/identity-manager/WorldIDIdentityManagerOwnershipManagement.t.sol:WorldIDIdentityManagerOwnershipManagement
[PASS] testCannotRenounceOwnershipIfNotOwner(address) (runs: 256, : 48782, ~: 48782)
[PASS] testCannotTransferOwnerIfNotOwner(address,address) (runs: 256, : 49177, ~: 49177)
[PASS] testHasOwner() (gas: 21854)
[PASS] testRenounceOwnership() (gas: 25314)
[PASS] testTransferOwner(address) (runs: 256, : 37839, ~: 37839)
Test result: ok. 5 passed; 0 failed; finished in 411.44ms
```

```
Running 5 tests for src/test/router/WorldIDRouterUninit.t.sol:WorldIDRouterUninit
[PASS] testCannotAddGroupWhileUninit(uint256,address) (runs: 256, : 1538875, ~: 1538875)
[PASS] testCannotDisableGroupWhileUninit(uint256) (runs: 256, : 1538593, ~: 1538593)
[PASS] testCannotGetGroupCountWhileUninit() (gas: 1538294)

[PASS] testCannotGetRouteForWhileUninit(uint256) (runs: 256, : 1538549, ~: 1538549)
[PASS] testCannotUpdateGroupWhileUninit(uint256,address) (runs: 256, : 1538966, ~: 1538966)
Test result: ok. 5 passed; 0 failed; finished in 705.28ms

Running 7 tests for src/test/identity-manager/WorldIDIdentityManagerStateBridge.t.sol:WorldIDIdentityManagerStateBridge
[PASS] testCanDisableStateBridgeFunctionality() (gas: 5406712)
[PASS] testCanEnableStateBridgeIfDisabled() (gas: 3370092)
[PASS] testCanUpgradestateBridge(address) (runs: 256, : 24107, ~: 24107)
[PASS] testCannotDisableStateBridgeIfAlreadyDisabled() (gas: 23999)
[PASS] testCannotEnableStateBridgeIfAlreadyEnabled() (gas: 18776)
[PASS] testCannotUpdateStateBridgeAsNonOwner(address) (runs: 256, : 47185, ~: 47185)
[PASS] testCannotUpgradeStateBridgeToZeroAddress() (gas: 17008)
Test result: ok. 7 passed; 0 failed; finished in 201.28ms

Running 4 tests for src/test/router/WorldIDRouterUpgrade.t.sol:WorldIDRouterUpgrade
[PASS] testCanUpgradeImplementationWithCall() (gas: 1535565)
[PASS] testCanUpgradeImplementationWithoutCall() (gas: 1504686)
[PASS] testCannotUpgradeUnlessManager(address) (runs: 256, : 1529864, ~: 1529864)
[PASS] testCannotUpgradeWithoutProxy(address) (runs: 256, : 1493187, ~: 1493187)
Test result: ok. 4 passed; 0 failed; finished in 229.40ms

Running 2 tests for
↪  src/test/identity-manager/WorldIDIdentityManagerSemaphoreVerification.t.sol:WorldIDIdentityManagerSemaphoreValidation
[PASS] testProofVerificationWithCorrectInputs(uint8,uint256,uint256,uint256,uint256[8]) (runs: 256, : 3484575, ~:
↪  3484575)
[PASS] testProofVerificationWithInorrectProof(uint8,uint256,uint256,uint256,uint256[8]) (runs: 256, : 3468459, ~:
↪  3468459)
Test result: ok. 2 passed; 0 failed; finished in 3.15s

Running 3 tests for src/test/router/WorldIDRouterStateBridge.t.sol:WorldIDRouterStateBridge
[PASS] testCanAddStateBridgeAsGroup(uint8,address) (runs: 256, : 2879543, ~: 1495851)
[PASS] testCanProxyVerifyProofForStateBridge(uint8,uint256,uint256,uint256,uint256,uint256[8]) (runs: 256, : 2381268,
↪  ~: 796816)
[PASS] testCanUpdateStateBridgeAsGroup(uint8) (runs: 256, : 2610789, ~: 1174729)
Test result: ok. 3 passed; 0 failed; finished in 14.12s

Running 17 tests for src/test/router/WorldIDRouterRouting.t.sol:WorldIDRouterRouting
[PASS] testCanAddGroup(address) (runs: 256, : 64062, ~: 64062)
[PASS] testCanDisableGroup(uint8,address) (runs: 256, : 4615273, ~: 2253427)
[PASS] testCanGetRouteForValidGroup(uint8,address,address) (runs: 256, : 2239281, ~: 607870)
[PASS] testCanUpdateGroup(uint8,address) (runs: 256, : 2581350, ~: 569576)
[PASS] testCannotAddDuplicateGroup(address) (runs: 256, : 28609, ~: 28609)
[PASS] testCannotAddGroupUnlessNumbersSequential(uint256,address) (runs: 256, : 32161, ~: 32161)
[PASS] testCannotAddGroupUnlessOwner(address) (runs: 256, : 51521, ~: 51521)
[PASS] testCannotAddGroupUnlessViaProxy(address) (runs: 256, : 8906, ~: 8906)
[PASS] testCannotDisableGroupUnlessOwner(address) (runs: 256, : 49227, ~: 49227)
[PASS] testCannotDisableGroupUnlessViaProxy(uint256) (runs: 256, : 8863, ~: 8863)
[PASS] testCannotUpdateGroupUnlessOwner(address) (runs: 256, : 51517, ~: 51517)
[PASS] testCannotUpdateGroupUnlessViaProxy(uint256,address) (runs: 256, : 9079, ~: 9079)
[PASS] testShouldRevertOnDisabledGroup(uint8) (runs: 256, : 2673149, ~: 1069852)
[PASS] testShouldRevertOnDisablingNonexistentGroup(uint256) (runs: 256, : 31838, ~: 31838)
[PASS] testShouldRevertOnRouteRequestForMissingGroup(uint256) (runs: 256, : 29052, ~: 29052)
[PASS] testShouldRevertOnUpdatingNonexistentGroup(uint256,address) (runs: 256, : 31572, ~: 31572)
[PASS] testcannotGetRouteUnlessViaProxy(uint256) (runs: 256, : 8796, ~: 8796)
Test result: ok. 17 passed; 0 failed; finished in 13.75s

Running 8 tests for src/test/identity-manager/WorldIDIdentityManagerDataQuery.t.sol:WorldIDIdentityManagerDataQuery
[PASS] testCanGetLatestRoot(uint256) (runs: 256, : 3377890, ~: 3378746)
[PASS] testCanGetTreeDepth(uint8) (runs: 256, : 3381665, ~: 3381665)
[PASS] testCannotGetLatestRootIfNotViaProxy() (gas: 8720)
[PASS] testCannotQueryRootIfNotViaProxy() (gas: 11186)
```

```
[PASS] testQueryCurrentRoot(uint128) (runs: 256, : 3396215, ~: 3397071)
[PASS] testQueryExpiredRoot(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4461181, ~: 4452379)
[PASS] testQueryInvalidRoot(uint256) (runs: 256, : 52986, ~: 52986)
[PASS] testQueryOlderRoot(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4442717, ~: 4423219)
Test result: ok. 8 passed; 0 failed; finished in 14.81s

Running 10 tests for
↪  src/test/identity-manager/WorldIDIdentityManagerIdentityUpdate.t.sol:WorldIDIdentityManagerIdentityUpdate
[PASS] testCannotRegisterIdentitiesWithOutdatedRoot(uint256,uint256,uint128[],uint128[8]) (runs: 256, : 4377288, ~:
↪  4378799)
[PASS] testCannotRegisterIdentitiesWithUnreducedIdentities(uint128,uint256,uint128,uint128[],uint128[8],bool) (runs:
↪  256, : 4404991, ~: 4402297)
[PASS] testCannotUpdateIdentitiesAsNonManager(address,uint128[],uint128[8]) (runs: 256, : 245188, ~: 237648)
[PASS] testCannotUpdateIdentitiesIfNotViaProxy(uint128[],uint128[8]) (runs: 256, : 165693, ~: 153891)
[PASS] testCannotUpdateIdentitiesWithIncorrectInputs(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 4536744, ~:
↪  4524380)
[PASS] testCannotUpdateIdentitiesWithInvalidBatchSize(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 4544864, ~:
↪  4540009)
[PASS] testCannotUpdateIdentitiesWithUnreducedPostRoot(uint128,uint128[],uint128[8]) (runs: 256, : 239038, ~: 235796)
[PASS] testCannotUpdateIdentitiesWithUnreducedPreRoot(uint128,uint128[],uint128[8]) (runs: 256, : 227578, ~: 217498)
[PASS] testUpdateIdentitiesSelectsCorrectVerifier(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 5022942, ~:
↪  4999556)
[PASS] testUpdateIdentitiesWithCorrectInputs(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 4542749, ~: 4520284)
Test result: ok. 10 passed; 0 failed; finished in 14.81s

Running 17 tests for
↪  src/test/identity-manager/WorldIDIdentityManagerIdentityRegistration.t.sol:WorldIDIdentityManagerIdentityRegistration
[PASS] testCannotRegisterIdentitiesAsNonManager(address) (runs: 256, : 75237, ~: 75237)
[PASS] testCannotRegisterIdentitiesIfIdentitiesIncorrect(uint256) (runs: 256, : 5361425, ~: 5361425)
[PASS] testCannotRegisterIdentitiesIfNotViaProxy() (gas: 41017)
[PASS] testCannotRegisterIdentitiesIfPostRootIncorrect(uint256) (runs: 256, : 5337024, ~: 5337024)
[PASS] testCannotRegisterIdentitiesIfStartIndexIncorrect(uint32) (runs: 256, : 5337128, ~: 5337128)
[PASS] testCannotRegisterIdentitiesWithIncorrectInputs(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, :
↪  4400789, ~: 4390435)
[PASS] testCannotRegisterIdentitiesWithInvalidBatchSize(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, :
↪  4397390, ~: 4404074)
[PASS] testCannotRegisterIdentitiesWithInvalidIdentities(uint8,uint8) (runs: 256, : 150921, ~: 161264)
[PASS] testCannotRegisterIdentitiesWithOutdatedRoot(uint256,uint256) (runs: 256, : 3419979, ~: 3420835)
[PASS] testCannotRegisterIdentitiesWithUnreducedIdentities(uint128) (runs: 256, : 85385, ~: 85385)
[PASS] testCannotRegisterIdentitiesWithUnreducedPostRoot(uint128) (runs: 256, : 68134, ~: 68134)
[PASS] testCannotRegisterIdentitiesWithUnreducedPreRoot(uint128) (runs: 256, : 63503, ~: 63503)
[PASS] testCannotRegisterIdentitiesWithUnreducedStartIndex(uint256) (runs: 256, : 41564, ~: 41564)
[PASS] testRegisterIdentitiesSelectsCorrectVerifier(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4789172,
↪  ~: 4774604)
[PASS] testRegisterIdentitiesWithCorrectInputs(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4423799, ~:
↪  4416458)
[PASS] testRegisterIdentitiesWithCorrectInputsFromKnown() (gas: 5404561)
[PASS] testRegisterIdentitiesWithRunsOfZeroes(uint8,uint8) (runs: 256, : 4385219, ~: 4406340)
Test result: ok. 17 passed; 0 failed; finished in 12.83s
```

5.6.5 Tests Output: State Bridge

```
> make test
FOUNDRY_PROFILE=debug forge test
[] Compiling...
[] Compiling 1 files with 0.8.15
[] Solc 0.8.15 finished in 789.39ms
Compiler run successful (with warnings)
warning[6321]: Warning: Unnamed return variable can remain unassigned. Add an explicit return with value to all
 ↪  non-reverting code paths or name the variable.
  --> lib/contracts/contracts/lib/MerklePatriciaProof.sol:20:30:
   |
20 |     ) internal pure returns (bool) {
   |                              ^^^^


warning[5667]: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> src/mock/WorldIDIdentityManagerMock.sol:25:29:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |                             ^^^^^^^^^^^^


warning[2018]: Warning: Function state mutability can be restricted to pure
  --> src/mock/WorldIDIdentityManagerMock.sol:25:5:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |     ^ (Relevant source part starts here and spans across multiple lines).



Running 2 tests for src/test/PolygonWorldID.t.sol:PolygonWorldIDTest
[PASS] testCanGetTreeDepth(uint8) (runs: 256, : 7017617, ~: 7017617)
[PASS] testConstructorWithInvalidTreeDepth(uint8) (runs: 256, : 6333465, ~: 6333486)
Test result: ok. 2 passed; 0 failed; finished in 274.27ms

Running 7 tests for src/test/OpWorldID.t.sol:OpWorldIDTest
[PASS] testCanGetTreeDepth(uint8) (runs: 256, : 7028161, ~: 7028161)
[PASS] testConstructorWithInvalidTreeDepth(uint8) (runs: 256, : 6353495, ~: 6353514)
[PASS] test_expiredRoot_reverts(uint256,uint256) (runs: 256, : 181496, ~: 181496)
[PASS] test_onlyOwner_notMessenger_reverts(uint256) (runs: 256, : 28747, ~: 28747)
[PASS] test_onlyOwner_notOwner_reverts(uint256) (runs: 256, : 32443, ~: 32443)
[PASS] test_receiveVerifyInvalidRoot_reverts(uint256) (runs: 256, : 122050, ~: 123077)
[PASS] test_receiveVerifyRoot_succeeds(uint256) (runs: 256, : 117204, ~: 118402)
Test result: ok. 7 passed; 0 failed; finished in 309.51ms

Running 7 tests for src/test/StateBridge.t.sol:StateBridgeTest
[PASS] test_canSelectFork_succeeds() (gas: 5639)
[PASS] test_notOwner_transferOwnershipOptimism_reverts(address,address,bool) (runs: 256, : 13915, ~: 13915)
[PASS] test_notOwner_transferOwnership_reverts(address,address) (runs: 256, : 13756, ~: 13756)
[PASS] test_owner_transferOwnershipOptimism_succeeds(address,bool) (runs: 256, : 117769, ~: 117769)
[PASS] test_owner_transferOwnership_succeeds(address) (runs: 256, : 20726, ~: 20726)
[PASS] test_sendRootMultichain_reverts(uint256,address) (runs: 256, : 174298, ~: 174298)
[PASS] test_sendRootMultichain_succeeds(uint256) (runs: 256, : 174563, ~: 174563)
Test result: ok. 7 passed; 0 failed; finished in 197.79s
```

5.6.6 Tests Output: AirDrop

```
> make test
FOUNDRY_PROFILE=debug forge test
[] Compiling...
[] Compiling 14 files with 0.8.19
[] Solc 0.8.19 finished in 2.08s
Compiler run successful

Running 4 tests for src/test/WorldIDAirdrop.t.sol:WorldIDAirdropTest
[PASS] testCanClaim(uint256,uint256) (runs: 256, : 95382, ~: 95382)
[PASS] testCannotDoubleClaim(uint256,uint256) (runs: 256, : 100246, ~: 100246)
[PASS] testCannotUpdateAirdropAmountIfNotManager(address) (runs: 256, : 14977, ~: 14977)
[PASS] testUpdateAirdropAmount() (gas: 19462)
Test result: ok. 4 passed; 0 failed; finished in 40.18ms

Running 6 tests for src/test/WorldIDMultiAirdrop.t.sol:WorldIDMultiAirdropTest
[PASS] testCanClaim(uint256,uint256) (runs: 256, : 220864, ~: 220864)
[PASS] testCanCreateAirdrop() (gas: 139154)
[PASS] testCanUpdateAirdropDetails() (gas: 149988)
[PASS] testCannotClaimNonExistantAirdrop(uint256,uint256) (runs: 256, : 50340, ~: 50340)
[PASS] testCannotDoubleClaim(uint256,uint256) (runs: 256, : 223165, ~: 223165)
[PASS] testNonOwnerCannotUpdateAirdropDetails(address) (runs: 256, : 144235, ~: 144235)
Test result: ok. 6 passed; 0 failed; finished in 48.01ms
```

5.6.7 Code Coverage: WorldID

```
> forge coverage
```

The relevant output is presented below.

```
-----------------------------------------------|------------------|------------------|-----------------|-----------------|
| File                                         | % Lines          | % Statements     | % Branches      | % Funcs         |
|----------------------------------------------|------------------|------------------|-----------------|-----------------|
| src/WorldIDIdentityManagerImplV1.sol         | 91.82% (101/110) | 92.44% (110/119) | 81.25% (39/48)  | 96.67% (29/30)  |
| src/WorldIDRouterImplV1.sol                  | 97.37% (37/38)   | 97.67% (42/43)   | 100.00% (12/12) | 90.00% (9/10)   |
| src/abstract/WorldIDImpl.sol                 | 100.00% (0/0)    | 100.00% (0/0)    | 100.00% (0/0)   | 100.00% (1/1)   |
| src/data/VerifierLookupTable.sol             | 100.00% (10/10)  | 100.00% (10/10)  | 100.00% (4/4)   | 100.00% (5/5)   |
| src/utils/CheckInitialized.sol               | 100.00% (1/1)    | 100.00% (1/1)    | 100.00% (0/0)   | 100.00% (1/1)   |
| src/utils/SemaphoreTreeDepthValidator.sol    | 0.00% (0/3)      | 0.00% (0/3)      | 100.00% (0/0)   | 0.00% (0/1)     |
| src/utils/UnimplementedTreeVerifier.sol      | 0.00% (0/5)      | 0.00% (0/5)      | 100.00% (0/0)   | 0.00% (0/1)     |
| Total                                        | 89.22% (149/167) | 90.05% (163/181) | 85.93% (55/64)  | 91.83% (45/49)  |
```

5.6.8 Code Coverage: State Bridge

```
> forge coverage
```

The relevant output is presented below.

```
-----------------------------------------------|----------------|-----------------|----------------|----------------|
| File                                         | % Lines        | % Statements    | % Branches     | % Funcs        |
|----------------------------------------------|----------------|-----------------|----------------|----------------|
| src/OpWorldID.sol                            | 50.00% (1/2)   | 50.00% (1/2)    | 100.00% (0/0)  | 50.00% (1/2)   |
| src/PolygonWorldID.sol                       | 0.00% (0/3)    | 0.00% (0/4)     | 100.00% (0/0)  | 0.00% (0/2)    |
| src/StateBridge.sol                          | 0.00% (0/16)   | 0.00% (0/18)    | 0.00% (0/2)    | 0.00% (0/5)    |
| src/abstract/WorldIDBridge.sol               | 63.64% (14/22) | 63.64% (14/22)  | 50.00% (6/12)  | 42.86% (3/7)   |
| src/utils/SemaphoreTreeDepthValidator.sol    | 0.00% (0/3)    | 0.00% (0/3)     | 100.00% (0/0)  | 0.00% (0/1)    |
| Total                                        | 32.60% (15/46) | 30.61% (15/49)  | 42.85% (6/14)  | 23.52% (4/17)  |
```

5.6.9 Code Coverage: AirDrop

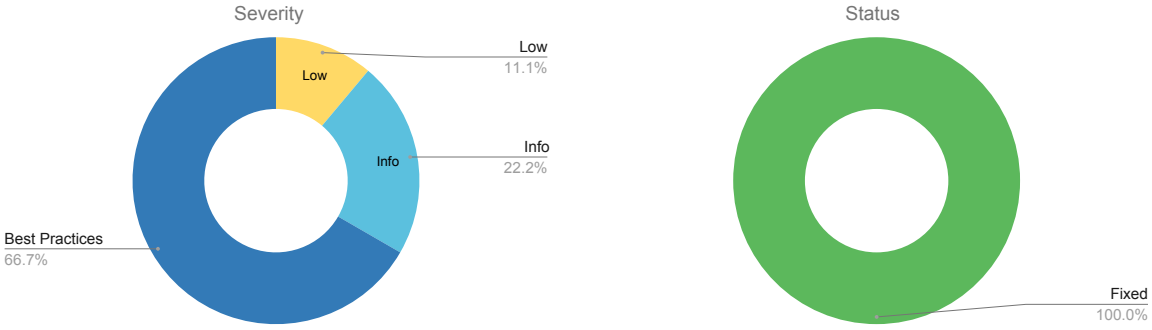
```
> forge coverage
```

The relevant output is presented below.

```
-------------------------------|-----------------|-----------------|-----------------|---------------|
| File                         | % Lines         | % Statements    | % Branches      | % Funcs       |
|------------------------------|-----------------|-----------------|-----------------|---------------|
| src/WorldIDAirdrop.sol       | 100.00% (7/7)   | 100.00% (8/8)   | 100.00% (4/4)   | 100.00% (2/2) |
| src/WorldIDMultiAirdrop.sol  | 100.00% (14/14) | 100.00% (17/17) | 100.00% (6/6)   | 100.00% (3/3) |
| Total                        | 100.00% (21/21) | 100.00% (25/25) | 100.00% (10/10) | 100.00% (5/5) |
```

6 WLD ERC20 Contract Audit Report

This section presents the security review performed by Nethermind on the WLD ERC20 Contract. Along this section, 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. 6.



(a) distribution of issues

(b) status of issues

Fig. 6: 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)

During the reaudit, the WorldCoin team has addressed all the identified issues, including implementing a modified inflation mechanism and reducing the operational and storage expenses associated with the protocol.

6.1 Technical Overview of the WLD ERC20 Contract

The audit consists of a smart contract for the WLD ERC20 Contract. 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 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;

```
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 if specific conditions are not 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 supply history array's size and 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 token's name, symbol, and decimals, set a new minter address, and prevent the owner from renouncing ownership.

Additionally, minter actions 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 comprehensive 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.

6.2 Documentation Evaluation: WLD ERC20 Contract

The code is well-commented, including the inflation mechanism.

6.3 Findings: WLD ERC20 Contract

6.3.1 [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 increase the entries in `supplyHistory` greatly. 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 cost the attacker more 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 make this 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.

6.3.2 [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.

6.3.3 [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, 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.

6.3.4 [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.

6.3.5 [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.

6.3.6 [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.

6.3.7 [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.

6.3.8 [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.

6.3.9 [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.

6.4 Test Suite Evaluation: WLD ERC20 Contract

6.4.1 Compilation Output: WLD ERC20 Contract

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

6.4.2 Tests Output: WLD ERC20 Contract

```
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
```

6.4.3 Code Coverage: WLD ERC20 Contract

```
> forge coverage
```

The relevant output is presented below.

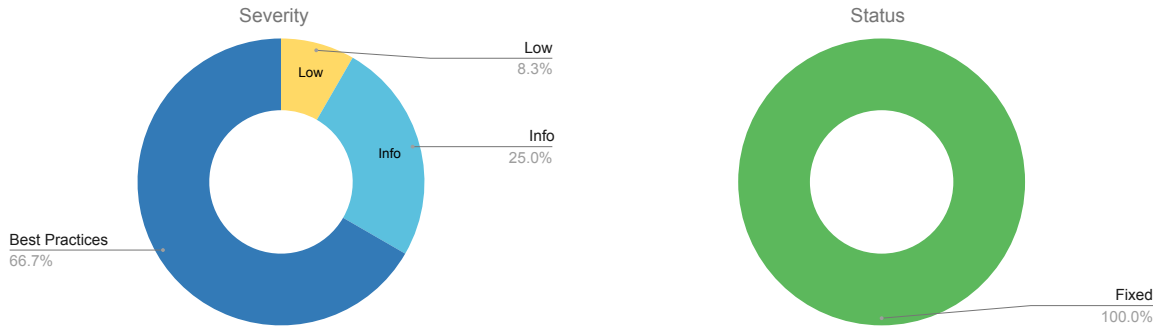
```
] Compiling...
[] Compiling 25 files with 0.8.19
[] Solc 0.8.19 finished in 1.91s
Compiler run successful
Analysing contracts...
Running tests...
| File        | % Lines        | % Statements   | % Branches    | % Funcs        |
|-------------|----------------|----------------|---------------|----------------|
| 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) |
```

7 Grants Contracts Audit Report

This section presents the security review performed by Nethermind on the WorldCoin Grants Contracts. The `Grants Contracts` facilitate recurring grant distributions of ERC20 tokens to specific participants. It is designed to distribute tokens periodically to individuals who meet specific criteria and can provide valid proof of eligibility. The contract enables the manager (deployer) to configure the grant by specifying parameters such as the `WorldID router`, `group ID`, `token` to be airdropped, `holder` (the address holding the tokens), and the `grant instance`. These parameters define the configuration of the recurring grant.

Participants can claim their tokens by invoking the `claim(...)` function with the required parameters. This function verifies the participant's eligibility by validating their proof against a provided `Merkle tree root`. If the claim is valid, the specified amount of tokens is transferred from the holder's address to the receiver's address. Internally, the `checkClaim(...)` function is used to validate the claim. The contract implements an access control mechanism that allows the manager to add or remove addresses from the list of allowed callers. Only the allowed callers can trigger the `claim(...)` function. To prevent double-signaling or multiple claims for the same grant, the contract maintains a mapping of `nullifier hashes` that have been previously used. If a `nullifier hash` is reused, an error is thrown to indicate an invalid claim.

Along the audit of these contracts, we report 12 points of attention, where 1 is classified as `Low`, 3 are classified as `Informational`, and 8 are classified as `Best Practices`. The issues are summarized in Fig. 7.



(a) distribution of issues

(b) status of issues

Fig. 7: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (3), **Best Practices** (8). **Distribution of status: Fixed** (12), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

7.1 Technical Overview of the Grants Contracts

The audit consists of the smart contracts `HourlyGrant.sol`, `RecurringGrantDrop.sol`, `WeeklyGrant.sol`, `MonthlyGrant.sol`, and the interface `IGrant.sol`. The interface `IGrant.sol` has the following methods.

```
interface IGrant {
    /// @notice Error in case the grant is invalid.
    error InvalidGrant();

    /// @notice Returns the current grant id.
    function getCurrentId() external view returns (uint256);

    /// @notice Returns the amount of tokens for a grant.
    /// @notice This may contain more complicated logic and is therefore not just a member variable.
    /// @param grantId The grant id to get the amount for.
    function getAmount(uint256 grantId) external view returns (uint256);

    /// @notice Checks whether a grant is valid.
    /// @param grantId The grant id to check.
    function checkValidity(uint256 grantId) external view;
}
```

The contract `HourlyGrant.sol` implements the interface `IGrant.sol` as follows.

```
contract HourlyGrant is IGrant {
    uint256 internal immutable offset;
    uint256 internal immutable amount;

    constructor(uint256 _offset, uint256 _amount) {
        offset = _offset;
        amount = _amount;
    }

    function getCurrentId() external view override returns (uint256) {
        return (block.timestamp - offset) / 3600;
    }

    function getAmount(uint256) external view override returns (uint256) {
        return amount;
    }

    function checkValidity(uint256 grantId) external view override{
        if (this.getCurrentId() != grantId) revert InvalidGrant();
    }
}
```

The contract `WeeklyGrant.sol` has a similar implementation to `HourlyGrant.sol`. The function `getCurrentId()` returns the number of weeks instead of the number of hours since the `offset`. The code is shown below.

```
contract WeeklyGrant is IGrant {
    uint256 internal immutable offset;
    uint256 internal immutable amount;

    constructor(uint256 _offset, uint256 _amount) {
        offset = _offset;
        amount = _amount;
    }

    function getCurrentId() external view override returns (uint256) {
        return (block.timestamp - offset) / (7*24*3600);
    }

    function getAmount(uint256) external view override returns (uint256) {
        return amount;
    }

    function checkValidity(uint256 grantId) external view override{
        if (this.getCurrentId() != grantId) revert InvalidGrant();
    }
}
```

The contract `MonthlyGrant.sol` also implements the interface `IGrant.sol` adding the function `calculateYearAndMonth(...)`, the state variables `startMonth` and `startYear`, and a slight change in the implementation of the function `getCurrentId()`. The contract is shown below.

```
contract MonthlyGrant is IGrant {
    uint256 internal immutable startMonth;
    uint256 internal immutable startYear;
    uint256 internal immutable amount;

    /// @param _startMonth The month of the first grant (1-12)
    /// @param _startYear The year of the first grant
    /// @param _amount The amount of tokens for each grant
    constructor(uint256 _startMonth, uint256 _startYear, uint256 _amount) {
        startMonth = _startMonth;
        startYear = _startYear;
        amount = _amount;
    }
```

```
    /// @notice Returns the current grant id starting from 0 (April 2023).
    function getCurrentId() external view override returns (uint256) {
        (uint256 year, uint256 month) = calculateYearAndMonth();
        return (year - startYear) * 12 + month - startMonth;
    }

    /// @notice Returns fixed amount of tokens for now.
    function getAmount(uint256) external view override returns (uint256) {
        // As stated in IGrant, this may contain more sophisticated logic in the future.
        return amount;
    }

    // @notice Anything that is not the current grant is invalid.
    function checkValidity(uint256 grantId) external view override {
        if (this.getCurrentId() != grantId) revert InvalidGrant();
    }

    /// @notice Returns the current year and month based on block.timestamp
    /// @notice Algorithm is taken from https://aa.usno.navy.mil/faq/JD_formula
    /// @return year The current year
    /// @return month The current month
    function calculateYearAndMonth() internal view returns (uint256, uint256) {
        uint256 d = block.timestamp / 86400 + 2440588;
        uint256 L = d + 68569;
        uint256 N = (4 * L) / 146097;
        L = L - (146097 * N + 3) / 4;
        uint256 year = (4000 * (L + 1)) / 1461001;
        L = L - (1461 * year) / 4 + 31;
        uint256 month = (80 * L) / 2447;
        // Removed day calculation:
        // uint256 day = L - (2447 * month) / 80;
        L = month / 11;
        month = month + 2 - 12 * L;
        year = 100 * (N - 49) + year + L;
        return (year, month);
    }
}
```

Finally, we have the contract `RecurringGrantDrop.sol`. The purpose of the `RecurringGrantDrop` contract is to facilitate recurring grant distributions of ERC20 tokens to specific participants. It implements a mechanism for distributing tokens periodically to individuals who meet certain criteria and can provide valid proof of eligibility. The contract allows the manager (deployer) to set up the grant by specifying the `WorldID router`, `group ID`, `token` to be airdropped, `holder` (address holding the tokens), and the `grant instance`. These parameters define the configuration of the recurring grant.

Participants can claim their tokens by calling the `claim(...)` function with the appropriate parameters. The function verifies the participant's eligibility by checking their proof against a provided `Merkle tree root`. If the claim is valid, the specified amount of tokens is transferred from the holder's address to the receiver's address. The `checkClaim(...)` function is used internally to validate the claim. The contract includes an access control mechanism that allows the manager to add or remove addresses from the list of allowed callers. Only the allowed callers can trigger the `claim(...)` function.

To prevent double-signaling or claiming the same grant multiple times, the contract maintains a mapping of `nullifier hashes` that have been used already. If a `nullifier hash` is reused, an error is thrown to indicate an invalid claim. The contract emits events to provide information about successful grant claims and updates to the grant instance.

The contract has the following imports.

```
import {ERC20} from "solmate/tokens/ERC20.sol";
import {SafeTransferLib} from "solmate/utils/SafeTransferLib.sol";
import { IGrant } from './IGrant.sol';
import {IWorldID} from "world-id-contracts/interfaces/IWorldID.sol";
import {IWorldIDGroups} from "world-id-contracts/interfaces/IWorldIDGroups.sol";
import {ByteHasher} from "world-id-contracts/libraries/ByteHasher.sol";
```

The contract also has the following standard errors and emits the following events.

```
/// @notice Thrown when restricted functions are called by not allowed addresses
error Unauthorized();

/// @notice Thrown when attempting to reuse a nullifier
error InvalidNullifier();

/// @notice Emitted when a grant is successfully claimed
/// @param receiver The address that received the tokens
event GrantClaimed(uint256 grantId, address receiver);

/// @notice Emitted when the grant is changed
/// @param grant The new grant instance
event GrantUpdated(IGrant grant);
```

The contract `RecurringGrantDrop.sol` has the following storage variables.

```
/// @dev The WorldID router instance that will be used for managing groups and verifying proofs
IWorldIDGroups internal immutable worldIdRouter;

/// @dev The World ID group whose participants can claim this airdrop
uint256 internal immutable groupId;

/// @notice The ERC20 token airdropped
ERC20 public immutable token;

/// @notice The address that holds the tokens that are being airdropped
/// @dev Make sure the holder has approved spending for this contract!
address public immutable holder;

/// @notice The address that manages this airdrop
address public immutable manager = msg.sender;

/// @notice The grant instance used
IGrant public grant;

/// @dev Whether a nullifier hash has been used already. Used to prevent double-signaling
mapping(uint256 => bool) internal nullifierHashes;

/// @dev Allowed addresses to call `claim`
mapping(address => bool) internal allowedCallers;
```

Finally, the contract implements the following functions.

- `constructor(IWorldIDGroups _worldIdRouter, uint256 _groupId, ERC20 _token, address _holder, IGrant _grant)`: Set up the contract operation;

- `claim(uint256 grantId, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof)`: Claims the airdrop by verifying the proof and transferring tokens to the receiver's address;

- `checkClaim(uint256 grantId, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof)`: Checks the validity of a claim by verifying the proof and grant validity;

- `addAllowedCaller(address _caller)`: Adds an address to the list of allowed callers;

- `removeAllowedCaller(address _caller)`: Removes an address from the list of allowed callers;

- `setGrant(IGrant _grant)`: Updates the grant.

The functions are reproduced below.

```
constructor(IWorldIDGroups _worldIdRouter, uint256 _groupId, ERC20 _token, address _holder, IGrant _grant)
{
    worldIdRouter = _worldIdRouter;
    groupId = _groupId;
    token = _token;
    holder = _holder;
    grant = _grant;
}
```

```
function claim(uint256 grantId, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof)
    public
{
    if (!allowedCallers[msg.sender]) revert Unauthorized();
    checkClaim(grantId, receiver, root, nullifierHash, proof);
    nullifierHashes[nullifierHash] = true;
    SafeTransferLib.safeTransferFrom(token, holder, receiver, grant.getAmount(grantId));
    emit GrantClaimed(grantId, receiver);
}

function checkClaim(uint256 grantId, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof)
    public
{
    if (nullifierHashes[nullifierHash]) revert InvalidNullifier();
    grant.checkValidity(grantId);
    worldIdRouter.verifyProof( groupId, root, abi.encodePacked(receiver).hashToField(),nullifierHash, grantId, proof );
}

function addAllowedCaller(address _caller) public {
    if (msg.sender != manager) revert Unauthorized();
    allowedCallers[_caller] = true;
}

function removeAllowedCaller(address _caller) public {
    if (msg.sender != manager) revert Unauthorized();
    allowedCallers[_caller] = false;
}

function setGrant(IGrant _grant) public {
    if (msg.sender != manager) revert Unauthorized();
    grant = _grant;
    emit GrantUpdated(_grant);
}
```

7.2 Documentation Evaluation: Grants Contracts

The documentation of the Grants Contract is presented in inline comments over the code, also the README.md file that points to other resources. We consider the documentation sufficient for performing this audit.

7.3 Findings: Grants Contracts

7.3.1 [Low] Solmate's `safeTransferFrom(...)` may silently fail if the specified token has no code

File(s): `src/RecurringGrantDrop.sol`

Description: Solmate's library is gas optimized and inspired by OpenZeppelin libraries. The `safeTransferFrom()` function implementation relies on inline assembly and on the call low-level function to be more gas efficient. The developer mentioned in the code that none of the functions in this library check that a token has code at all! That responsibility is delegated to the caller. The return value will be `true` if the `call` is done against a non-existent account. Suppose that a `RecurrentGrantDrop` contract is initialized with a wrong token address with no code. The `claim()` would silently fail and emit the `GrantClaimed()` event without reverting.

Recommendation(s): Consider either checking that the `token` has code or use Openzeppelin's `SafeERC20` library to transfer of the air-dropped tokens.

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.2 [INFO] Missing input validation in `MonthlyGrant.constructor(...)`

File(s): `src/MonthlyGrant.sol`

Description: The `MonthlyGrant.constructor()` does not check the validity of the input parameters `_startMonth` and `_startYear`. Those parameters define the month and year of the grant's payment. These parameters are provided in the contract's constructor and are stored immutably. In the `getCurrentId()` function, the contract calculates the current `grantId` based on the current month and year compared to the start month and year. This is done to determine the number of grants that have occurred since the beginning. The ID calculation is based on the difference in months between the current month and the start month, multiplied by the number of months in a year plus the difference in years.

In the `checkValidity()` function, the contract checks if the provided ID matches the current grant ID. This is done by comparing the provided ID with the current ID obtained from the `getCurrentId()` function. If the provided ID differs from the current ID, an exception indicates that the grant is invalid.

The `startMonth` and `startYear` parameters are used to control and track the progression of the grant over time. They allow the contract to calculate the current grant ID based on the difference in months and years since the grant's start and validate a provided ID against the current ID. The code is reproduced below.

```
constructor(uint256 _startMonth, uint256 _startYear, uint256 _amount) {
  ////////////////////////////////////////////////////////////////////////
  // @audit Missing input validation for "_startMonth" and "_startYear"
  ////////////////////////////////////////////////////////////////////////
  startMonth = _startMonth;
  startYear = _startYear;
  amount = _amount;
}

function getCurrentId() external view override returns (uint256) {
    (uint256 year, uint256 month) = calculateYearAndMonth();
    return (year - startYear) * 12 + month - startMonth;
}

```

Recommendation(s): Consider checking that `_startMonth` is in between the range of `1` to `12`. Also, check the validity of the `_startYear`.

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.3 [INFO] Possible underflow when retrieving current `grantId`

File(s): src/HourlyGrant.sol, WeeklyGrant.sol, MontlyGrant.sol

Description: The `getCurrentId()` function returns the current `grantId`. It does so by calculating the elapsed time since the start of the grant. However, if the grant is created upstream and inactive, the calculation will revert without any explicit error message. The revert would occur since the `offset` parameter would be higher than the current `block.timestamp`. Below we present those functions for the contracts `HourlyGrant`, `WeeklyGrant`, and `MontlyGrant`.

```
function HourlyGrant.getCurrentId() external view override returns (uint256) {
        return (block.timestamp - offset) / 3600;
    }
```

```
    function WeeklyGrant.getCurrentId() external view override returns (uint256) {
        return (block.timestamp - offset) / (7*24*3600);
    }
```

```
    function MontlyGrant.getCurrentId() external view override returns (uint256) {
        (uint256 year, uint256 month) = calculateYearAndMonth();
        return (year - startYear) * 12 + month - startMonth;
    }
```

Recommendation(s): Consider reverting with a specific error when the `offset` (or the current number of months for `MonthlyGrant`) is lower than the current `blockchain timestamp` (or current `month`).

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.4 [Info] Ensure that you are the owner after deployment the contract `RecurringGrantDrop`

File(s): src/RecurringGrantDrop.sol

Description: The manager of the airdrop is an immutable variable initialized with `msg.sender`. Before interacting with the contract, ensure that you are the contract owner. The variable is reproduced below.

```
/// @notice The address that manages this airdrop
address public immutable manager = msg.sender;
```

Recommendation(s): Ensure that you are the contract owner before interacting with it.

Status: Fixed

Update from the client: Fixed based on this suggestion and the adoption of `Ownable2Step` as we can check here.

7.3.5 [Best Practice] Confusing date description in `MonthlyGrant.getCurrentId()`

File(s): src/MonthlyGrant.sol

Description: The `getCurrentId()` function's documentation of `MonthlyGrant` contract mentions that the grant starting date is set to `April 2023`. However, there is no such initialization logic in the contract code.

```
/// @notice Returns the current grant id starting from 0 (April 2023).
function getCurrentId() external view override returns (uint256) {
    (uint256 year, uint256 month) = calculateYearAndMonth();
    return (year - startYear) * 12 + month - startMonth;
}
```

Recommendation(s): Consider updating the comment by removing the explicit date or putting more explanation to avoid confusion.

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.6 [Best Practice] Defining Storage Variables after Events and Errors

File(s): src/RecurringGrantDrop.sol

Description: One good practice is following the order of Layout of Solidity lang documentation. Specifically, it is recommended to place storage variables before events and error declarations in the contract.

Recommendation(s): To adhere to best practices, consider placing the storage variables declaration before the events and errors section.

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.7 [Best Practice] Function that can have `external` visibility

File(s): src/RecurringGrantDrop.sol

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

```
setGrant(IGrant _grant)
removeAllowedCaller(address _caller)
addAllowedCaller(address _caller)
claim(uint256 grantId, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof)
```

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

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.8 [Best Practice] Missing event during sensitive storage variable update

File(s): src/RecurringGrantDrop.sol

Description: The `RecurringGrantDrop` contract initializes sensitive contract variables within the constructor. Not any event is emitted to notify of the state change of the contract.

Recommendation(s): Consider emitting an event when creating a new `RecurringGrandDrop` contract.

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.9 [Best Practice] Use of Solidity's date and time units instead of calculations

File(s): src/WeeklyGrant.sol , src/HourlyGrant.sol

Description: The `getCurrentId()` function calculates the current grant id based on the difference between the current block's timestamp and the offset value. It divides the difference by the number of seconds a week (`7 * 24 * 3600`) or an hour (`3600`). However, Solidity provides the native date and time units to improve the readability and maintenance of code.

```
function getCurrentId() external view override returns (uint256) {
    return (block.timestamp - offset) / 3600; // @audit: consider using `1 hours` instead of `3600`.
    }
```

Recommendation(s): Consider using a more readable format for the calculation, such as `7 days` instead of `7*24*3600` in `WeeklyGrant.sol`, and `1 hours` instead of `3600` in `HourlyGrant.sol`.

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.10 [Best Practices] `RecurringGrantDrop.addAllowedCaller(...)` not checking `_caller` for `address(0x0)`

File(s): src/RecurringGrantDrop.sol

Description: The functions `RecurringGrantDrop.addAllowedCaller(...)` and `RecurringGrantDrop.removeAllowedCaller(...)` do not check `_caller` for `address(0x0)`. The function are reproduced below.

```
/// @notice Add a caller to the list of allowed callers
/// @param _caller The address to add
function addAllowedCaller(address _caller) public {
    if (msg.sender != manager) revert Unauthorized();
    ////////////////////////////////////////////////////////
    // @audit Not checking "_caller" for "address(0x0)"
    ////////////////////////////////////////////////////////
    allowedCallers[_caller] = true;
}
```

Recommendation(s): This function has access control and can only be called by authorized addresses. However, checking for `address(0x0)` represents a small cost and can prevent broken invariants.

Status: Fixed

Update from the client: Fixed in this pull request.

7.3.11 [Best Practices] Function `checkClaim(...)` may have `internal` visibility

File(s): src/RecurringGrantDrop.sol

Description: The function `RecurringGrantDrop.checkClaim(...)` seems to have `internal` visibility (instead of `public`) since it is only called by the function `claim(...)`. The code is reproduced below.

```
function claim(uint256 grantId, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof)
    public
{
    if (!allowedCallers[msg.sender]) revert Unauthorized();

    ////////////////////////////////////////////////////////////////
    // @audit "checkClaim(...)" is only called by "claim(...)"
    ////////////////////////////////////////////////////////////////
    checkClaim(grantId, receiver, root, nullifierHash, proof);

    nullifierHashes[nullifierHash] = true;

    SafeTransferLib.safeTransferFrom(token, holder, receiver, grant.getAmount(grantId));

    emit GrantClaimed(grantId, receiver);
}
```

Recommendation(s): Evaluate if this change can be made. Maybe you have plans to use this function in other modules in the future.

Status: Fixed

Update from the client: We are using this function as a ”pre-flight check” from the caller only to check the proof-related logic to be correct (not the additional checks). It would be beneficial to keep it like this to avoid changes in the (off-chain) caller.

Update from Nethermind: We agree with the client.

7.3.12 [Best Practices] Functions `addAllowedCaller(...)` and `removeAllowedCaller(...)` not emitting events

File(s): src/RecurringGrantDrop.sol

Description: The functions `RecurringGrantDrop.addAllowedCaller(...)` and `RecurringGrantDrop.removeAllowedCaller(...)` do not emit events. The function is reproduced below.

```
/// @notice Add a caller to the list of allowed callers
/// @param _caller The address to add
function addAllowedCaller(address _caller) public {
    if (msg.sender != manager) revert Unauthorized();
    allowedCallers[_caller] = true;
}
```

```
/// @notice Remove a caller from the list of allowed callers
/// @param _caller The address to remove
function removeAllowedCaller(address _caller) public {
    if (msg.sender != manager) revert Unauthorized();
    allowedCallers[_caller] = false;
}
```

Recommendation(s): Consider emiting events with the `_caller` in these functions. Those events can be important if you decide to implement monitoring tools for smart contracts.

Status: Fixed

Update from the client: Fixed in this pull request.

7.4 Test Suite Evaluation: Grants Contracts

7.4.1 Compilation Output: Grants Contracts

```
> make
forge install && npm install

added 461 packages, and audited 462 packages in 14s

83 packages are looking for funding
  run `npm fund` for details

15 vulnerabilities (10 moderate, 4 high, 1 critical)

To address issues that do not require attention, run:
  npm audit fix

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.
npm notice
npm notice New minor version of npm available! 9.3.1 -> 9.8.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v9.8.0
npm notice Run npm install -g npm@9.8.0 to update!
npm notice
forge build
[] Compiling...
[] Compiling 31 files with 0.8.19
[] Solc 0.8.19 finished in 2.88s
Compiler run successful
```

7.4.2 Tests Output: Grants Contracts

```
forge test
Running 2 tests for src/test/MonthlyGrant.t.sol:MonthlyGrantTest
[PASS] testInitialMonth() (gas: 12779)
[PASS] testMoreMonths() (gas: 373815)
Test result: ok. 2 passed; 0 failed; finished in 102.42ms

Running 6 tests for src/test/RecurringGrantDrop.t.sol:RecurringGrantDropTest
[PASS] testCanClaim(uint256,uint256) (runs: 256, : 114754, ~: 114754)
[PASS] testCannotClaimFuture(uint256,uint256) (runs: 256, : 55095, ~: 55095)
[PASS] testCannotClaimPast(uint256,uint256) (runs: 256, : 55219, ~: 55219)
[PASS] testCannotDoubleClaim(uint256,uint256) (runs: 256, : 120536, ~: 120536)
[PASS] testCannotUpdateGrantIfNotManager(address) (runs: 256, : 260716, ~: 260716)
[PASS] testUpdateGrant() (gas: 263694)
Test result: ok. 6 passed; 0 failed; finished in 101.26ms
```

7.4.3 Code Coverage: Grants Contracts

```
> forge coverage
```

The relevant output is presented below.

```
[] Compiling...
[] Compiling 31 files with 0.8.19
[] Solc 0.8.19 finished in 1.99s
Compiler run successful
Analysing contracts...
Running tests...
| File                                               | % Lines         | % Statements    | % Branches    | % Funcs       |
|----------------------------------------------------|-----------------|-----------------|---------------|---------------|
| script/RecurringGrantDrop.s.sol                    | 0.00% (0/5)     | 0.00% (0/5)     | 0.00% (0/2)   | 0.00% (0/1)   |
| script/utils/SetAllowanceERC20.s.sol               | 0.00% (0/3)     | 0.00% (0/3)     | 100.00% (0/0) | 0.00% (0/1)   |
| src/HourlyGrant.sol                                | 0.00% (0/3)     | 0.00% (0/4)     | 0.00% (0/2)   | 0.00% (0/3)   |
| src/MonthlyGrant.sol                               | 100.00% (15/15) | 100.00% (22/22) | 100.00% (2/2) | 100.00% (4/4) |
| src/RecurringGrantDrop.sol                         | 73.33% (11/15)  | 63.16% (12/19)  | 50.00% (5/10) | 40.00% (2/5)  |
| src/WeeklyGrant.sol                                | 0.00% (0/3)     | 0.00% (0/4)     | 0.00% (0/2)   | 0.00% (0/3)   |
| src/test/mock/TestERC20.sol                        | 0.00% (0/1)     | 0.00% (0/1)     | 100.00% (0/0) | 0.00% (0/1)   |
| src/test/mock/WorldIDIdentityManagerRouterMock.sol | 100.00% (1/1)   | 100.00% (1/1)   | 100.00% (0/0) | 100.00% (1/1) |
| Total                                              | 58.70% (27/46)  | 59.32% (35/59)  | 38.89% (7/18) | 36.84% (7/19) |
```

8 Vesting Wallet Audit Report

This section presents the security review performed by Nethermind on the WorldCoin Vesting Wallet Contracts. The `VestingWallet` smart contract handles the vesting of the WLD ERC20 Token for a specific beneficiary through a linear vesting schedule. It enables custody of multiple tokens, and once transferred to the contract, the tokens will follow the vesting schedule as if they were locked from the beginning. The vesting schedule is customizable and is determined by the `start` and `duration` variables. The `start` variable represents the start timestamp of the vesting period, and the `duration` variable represents the duration of the vesting period in seconds. By setting the `duration` to 0, the contract can also act as an asset timelock, holding tokens for the beneficiary until a specified time.

The beneficiary is controlled through the `{Ownable}` mechanism, allowing them to assign the unreleased portion of the assets to another party. To determine the amount of tokens that are currently releasable by the beneficiary, the contract provides the `releasable(...)` function. This function calculates the vested amount based on the vesting schedule and subtracts the amount of already released tokens.

To release the tokens that have vested, the beneficiary can call the `release(...)` function, which will emit an `{ERC20Released}` event and transfer the released tokens to the contract owner (the beneficiary) using the `{SafeERC20}` utility.

The actual calculation of the vested amount is performed by the `vestedAmount` function, which uses a linear vesting curve. This function takes a token address and a timestamp as input and returns the amount of vested tokens at that specific time. The vested amount increases linearly over time until the end of the vesting period is reached.

To ensure proper contract deployment, an error check is included to verify that a valid beneficiary address is provided.

No issues were found in the implementation of the `VestingWallet.sol` and the `WLD.sol`.

8.1 Technical Overview of the Vesting Wallet Contracts

The audit consists of the smart contracts `VestingWallet.sol` and `WLD.sol`.

8.1.1 VestingWallet.sol

The contract has the following imports.

```
import {IERC20} from "openzeppelin/token/ERC20/IERC20.sol";
import {SafeERC20} from "openzeppelin/token/ERC20/utils/SafeERC20.sol";
import {Ownable2Step} from "openzeppelin/access/Ownable2Step.sol";
import {Ownable} from "openzeppelin/access/Ownable.sol";
```

The contract defines the following event.

```
event ERC20Released(address indexed token, uint256 amount);
```

The contract defines the following standard error.

```
error VestingWalletInvalidBeneficiary(address beneficiary);
```

The contract has the following storage variables.

```
mapping(address => uint256) public released;
uint64 public immutable start;
uint64 public immutable duration;
uint64 public immutable end;
```

The contract has the following functions.

- `constructor`: This is the constructor function called when the contract is deployed. It takes three parameters: `beneficiaryAddress`, `startTimestamp`, and `durationSeconds`. The `beneficiaryAddress` is the address of the beneficiary who will receive the vested tokens. The `startTimestamp` represents the start time of the vesting period, and `durationSeconds` is the duration of the vesting period in seconds. It sets up the vesting schedule for the beneficiary.

- `releasable`: This function takes an ERC20 token contract address as input and returns the amount of tokens that are currently releasable by the beneficiary. It calculates the vested amount based on the vesting schedule and subtracts the amount of already released tokens.

- `release`: This function allows the beneficiary to release the already vested tokens. It calculates the amount of tokens that are releasable using the `releasable` function, increments the `released` mapping for the specific token, emits an `ERC20Released` event and then transfers the released tokens to the contract owner (the beneficiary) using the `SafeERC20.safeTransfer` function.

- `vestedAmount`: This function calculates the amount of tokens that have already vested at a given timestamp using a linear vesting curve. It takes a token address and a timestamp as input and returns the amount of vested tokens at that specific time. The vested amount increases linearly over time until the end of the vesting period is reached.

8.1.2 WLD.sol

The WLD.sol smart contract is an ERC20 token contract for Worldcoin's WLD token. The contract has the following imports.

```
import {Ownable} from "openzeppelin/access/Ownable.sol";
import {Ownable2Step} from "openzeppelin/access/Ownable2Step.sol";
import {ERC20} from "openzeppelin/token/ERC20/ERC20.sol";
```

The contract has the following storage variables.

```
uint256 constant public INITIAL_SUPPLY_CAP = 10_000_000_000 * (10**18);
uint256 constant public WAD_ONE = 10**18;

/// @notice Has the initial mint been done?
bool public initialMintDone;

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

/// @notice Inflation parameters, formula in _mint @dev description
uint256 immutable public inflationUnlockTime;
uint256 immutable public inflationCapPeriod;
uint256 immutable public inflationCapWad;

/// @notice Inflation cap state variables
uint256 public currentPeriodEnd;
uint256 public currentPeriodSupplyCap;
```

The contract defines the following events.

```
/// @notice Emitted when constructing the contract
event TokenUpdated(
    address newToken,
    string name,
    string symbol,
    address[] existingHolders,
    uint256[] existingsAmounts,
    uint256 inflationCapPeriod,
    uint256 inflationCapWad,
    uint256 inflationLockPeriod
);

/// @notice Emitted when minting tokens. Can be emited only once.
event TokensMinted(
    address minter,
    address[] newHolders,
    uint256[] newAmounts
);

/// @notice Emitted when inflation tokens are minted, after the initial mint.
event InflationTokensMinted(
    address minter,
    address to,
    uint256 amount
);
```

The contract has the following functions.

- constructor: The constructor is called when deploying the contract. It takes several parameters, including existing holders and their token amounts, the name and symbol of the token, inflation parameters, and an inflation lock period. The contract has these initial values and a total supply cap of 10 billion tokens.

- mintOnce: This function allows the contract owner to perform a one-time mint of new tokens up to the INITIAL_SUPPLY_CAP. **It can only be called once and marks the initial mint as done**.

- setMinter: This function is used by the owner to set the address of the inflation minter, which can mint new tokens up to the inflation cap after the inflationUnlockTime has passed.

- mintInflation: The minter uses this function to mint new tokens and assign them to a target address. It enforces inflation checks, such as ensuring that the current time is after the inflation lock-in period, calculating the inflation period's supply cap, and preventing excessive inflation. It allows for controlled inflation of tokens.

8.2 Findings: Vesting Wallet Contract

No issues were found in the implementation of the `VestingWallet.sol` and the `WLD.sol`.

8.3 Documentation Evaluation: Vesting Wallet Contract

The documentation of the Grants Contract is presented in inline comments over the code.

8.4 Test Suite Evaluation: Vesting Wallet Contract

8.4.1 Compilation Output: Vesting Wallet Contract

```
> forge compile
forge test
Missing dependencies found. Installing now...

[] Compiling...
[] Compiling 33 files with 0.8.19
[] Solc 0.8.19 finished in 3.45s
Compiler run successful (with warnings)
warning[2072]: Warning: Unused local variable.
   --> test/WLD.t.sol:227:9:
    |
227 |         WLD token = new WLD(holders, amounts, _symbol, _name, _inflationCapPeriod, _inflationCapWad,
 ↪  _mintLockInPeriod);
    |         ^^^^^^^^^
```

8.4.2 Tests Output: Vesting Wallet Contract

```
forge test
Running 5 tests for test/VestingWallet.t.sol:VestingWalletTest
[PASS] testERC20OwnershipTransfer() (gas: 1224857)
[PASS] testERC20VestingExecution() (gas: 1984792)
[PASS] testERC20VestingGetters() (gas: 1447715)
[PASS] testParameterGetters() (gas: 14828)
[PASS] testZeroBeneficiary() (gas: 45170)
Test result: ok. 5 passed; 0 failed; finished in 17.89ms

Running 16 tests for test/WLD.t.sol:WLDTest
[PASS] testCanMintOnce() (gas: 108275)
[PASS] testDecimals() (gas: 5470)
[PASS] testFailCanMintOnlyOnce() (gas: 7720)
[PASS] testFailCapInitial() (gas: 211463)
[PASS] testFailCapSecond() (gas: 1187240)
[PASS] testFailOnlyOnceMinter(address) (runs: 256, : 27976, ~: 27976)
[PASS] testInflationCap() (gas: 204489)
[PASS] testInitialDistributionHappens() (gas: 13879)
[PASS] testInitialDistributionRestricted(address) (runs: 256, : 11005, ~: 11005)
[PASS] testLargeDistribution() (gas: 6656696)
[PASS] testMintAccessControl(address) (runs: 256, : 18574, ~: 18574)
[PASS] testMintsLockInPeriod() (gas: 101664)
[PASS] testSecondDistribution() (gas: 77206)
[PASS] testSecondDistributionAdditive() (gas: 117637)
[PASS] testSecondDistributionRestricted(address) (runs: 256, : 112492, ~: 112492)
[PASS] testSetMinterSucceeds(address) (runs: 256, : 19469, ~: 19469)
Test result: ok. 16 passed; 0 failed; finished in 35.14ms
```

8.4.3 Code Coverage: Vesting Wallet Contract

```
> forge coverage
```

The relevant output is presented below.

```
Analysing contracts...
Running tests...
| File                    | % Lines         | % Statements    | % Branches     | % Funcs      |
|-------------------------|-----------------|-----------------|----------------|--------------|
| script/DeployWLD.s.sol | 0.00% (0/13)    | 0.00% (0/13)    | 0.00% (0/2)    | 0.00% (0/1)  |
| src/VestingWallet.sol  | 90.91% (10/11) | 92.31% (12/13) | 75.00% (3/4)   | 66.67% (2/3) |
| src/WLD.sol            | 95.24% (20/21) | 96.00% (24/25) | 72.22% (13/18) | 75.00% (3/4) |
| Total                   | 66.67% (30/45) | 70.59% (36/51) | 66.67% (16/24) | 62.50% (5/8) |
```

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

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.